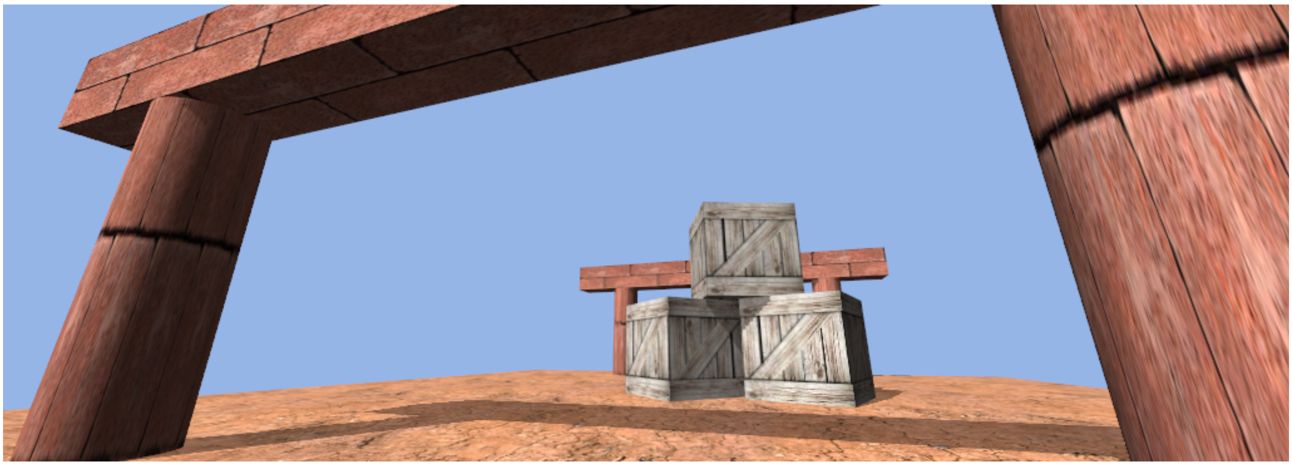
مقدمة عملية إلى برمجة الألعاب الإلكترونية

الإصدار الأول



ياسر جفال

ديسمبر 2014

3dgd-intro.blogspot.com

قائمة المواضيع

الوحدة الأولى: أساسيات بناء المشهد

سنتعرف في هذه الوحدة إن شاء الله على أهم الكائنات التي يحويها مشهد اللعبة ثلاثية الأبعاد، وأهم الخصائص المتعلقة بهذه الكائنات، وكذلك العلاقات التي تربط هذه الكائنات ببعضها البعض. فإذا كنت ممن يستخدمون برامج التصميم ثلاثية الأبعاد فلا شك أنك ستلاحظ شبها كبيرا بين تركيب المشهد في محرك Unity وبرنامج أو برامج التصميم التي تعرفها. إن كنت من هؤلاء فيمكنك تخطي الفصول الأولى من هذه الوحدة دون قلق من أن يفوتك شرح شيء مهم والقفز مباشرة إلى فصلها الأخير، وإن أردت متابعة قراءتها فلك ذلك. لن نتطرق في هذه الوحدة إلى أية مواضيع متقدمة في بناء المشهد، بل سنكتفي بالأساسيات التي تمكننا من الاستمرار في طريقنا نحو تعلّم فنون البرمجة الخاصة بالألعاب.

ما يتوقع منك عند الانتهاء من دراسة هذه الوحدة:

- معرفة كيفية بناء المشهد ثلاثي الأبعاد باستخدام الكائنات المختلفة
- استيعاب الخصائص المتعلقة بكائنات المشهد مثل الموقع والحجم والدوران
- فهم طبيعة الروابط التي تربط كائنات المشهد وكيف تؤثر على مظهرها وسلوكها
- التعرف على بعض الخصائص الأساسية المتعلقة بالتصيير مثل الإكساء والمواد والمظللات
- التعرف على أنواع الضوء الأساسية وكيفية استخدامها وأهم خصائصها
- استخدام الكاميرا في تصيير المشهد للاعب
- معرفة كيفية تغيير خصائص الكائنات خلال تشغيل اللعبة للحصول على التأثيرات المطلوبة

الفصل الأول: الأشكال الأساسية وخصائصها

بالإضافة إلى إمكانية استيراد نماذج ثلاثية الأبعاد من معظم برامج التصميم المعروفة، يحتوي محرك Unity أيضا على مجموعة من الكائنات (game objects) والتي تمثل أشكالا هندسية أساسية، مما يمكّنك من بناء مشهد بسيط والتفاعل معه. من الأمثلة على هذه الكائنات المكعب (cube) والكرة (sphere) واللوح (plane) والأسطوانة (cylinder). فباستطاعتك بناء المشهد بإضافة عدد غير محدد من هذه الأشكال والتغيير في خصائصها مثل الموقع (position) والحجم (يسمى أيضا القياس) (scale) والدوران (rotation).

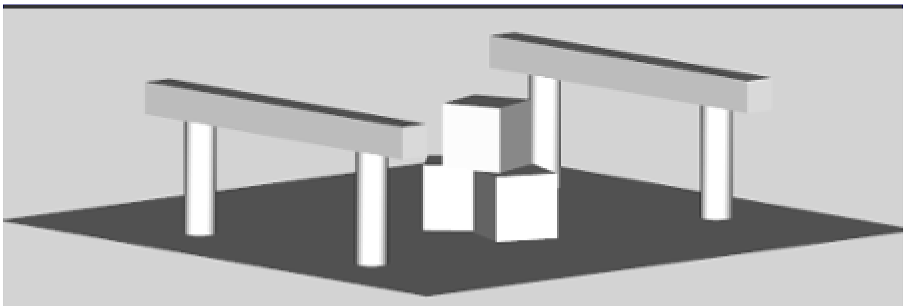
يمكنك إضافة هذه الأشكال بالدخول إلى قائمة Game Object ثم Create Other، ستجدها في القسم الثالث من القائمة ابتداء من Cube وانتهاء بـ Quad

بمجرد إضافة أي من هذه الكائنات إلى المشهد يمكنك أن تشاهده في هرمية المشهد (Hierarchy) وكذلك في نافذة المشهد (Scene)، إن لم تتمكن من رؤيته في نافذة المشهد يمكنك ببساطة اختياره من هرمية المشهد ومن ثم وضع المؤشر داخل نافذة المشهد والضغط على مفتاح F على لوحة المفاتيح للتركيز عليه.

ستلاحظ عند البدء في بناء أي مشهد أن هرمية المشهد تحتوي فقط على كائن واحد وهو Main Camera وهي الكاميرا المسؤولة بطبيعة الحال عن تصيير المشهد أثناء اللعب. بكلمات أخرى، هذه الكاميرا هي عين اللاعب على فضاء اللعبة.

لنجرب معا بناء مشهد سريع مستخدمين بعض كائنات الأشكال الأساسية. حاول أن تبني المشهد بنفسك بعد أن تطّلع على الشكل 1، إن وجدت صعوبة في ذلك يمكنك أن تقرأ الوصف في الفقرة التي تلي الشكل.

يمكنك تغيير موقع وحجم ودوران الأشكال من خلال المشهد مباشرة عن طريق اختيار الأداة المناسبة من الأزرار  الموجودة أعلى يسار نافذة Unity، كما يمكنك أيضا إدخال القيم مباشرة عن طريق المكوّن Transform والذي يظهر في نافذة الخصائص Inspector (الشكل 2)



الشكل 1: مشهد بسيط تم بناؤه باستخدام الأشكال الأساسية المتوفرة في محرك Unity

لبناء الشكل 1 علينا معرفة أنواع كائنات الأشكال الأساسية المضافة ومواقعها وقياساتها، إضافة إلى الدوران الخاص بكل منها، وذلك على المحاور المختلفة في الفضاء ثلاثي الأبعاد (x, y, z). قبل التطرق لخطوات بناء المشهد، علينا أن نعرف ما هي هذه المحاور الثلاث وماذا تمثل في الفضاء، ولنستخدم لذلك قاعدة اليد اليسرى. أشر بسبابة يدك اليسرى إلى الأمام، وبإصبع الوسطى إلى اليمين، وبالإبهام للأعلى. في هذه الحالة تشير السبابة إلى الاتجاه الموجب للمحور z، وتشير الوسطى إلى الاتجاه

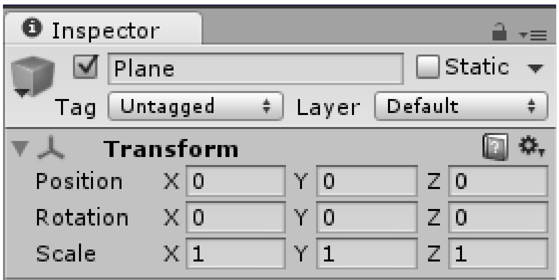
الموجب للمحور x، ويشير الإبهام إلى الاتجاه الموجب للمحور y. يمكن بناء المشهد فيالشكل 1 باستخدام الخطوات التالية:

1. لبناء الأرضية قم بإضافة لوح إلى المشهد، موقعه في نقطة الأصل (0, 0, 0) بينما حجمه واستدارته تبقى على قيمها الافتراضية وهي (1, 1, 1) بالنسبة للحجم و (0, 0, 0) بالنسبة للدوران. ببقائه على الحجم الأصلي، سيغطي اللوح مساحة 100 (10 * 10) وحدة مربعة على المستوى (x, z) والذي يمثل المساحة الأرضية بينما يمثل المحور y الارتفاع.

2. لبناء الأعمدة قم بإضافة أربع أسطوانات إلى المشهد بحيث تتوزع حول نقطة الأصل على شكل مستطيل. للحصول على التوزيع المطلوب يجب أن تأخذ الأعمدة أربع مواقع على المستوى (x, z) وهي (2, 3.5) و (-2, 3.5) و (2, -3.5) و (-2, -3.5) بينما ترتفع وحدة واحدة على المحور y (أي y = 1 لمواقع جميع الأعمدة) وذلك حتى يكون سطحها السفلي ملامسا للأرضية. لجعل الأعمدة أقل سماكة، قم بتغيير الحجم على المحورين x و z إلى 0.5 بالتالي يقل قطرها إلى نصف قيمته الأصلية.

3. لبناء العوارض فوق الأعمدة قم بإضافة مكعبين للمشهد في المواقع (0, 2, -3.5) و (0, 2, 3.5) ثم قم بتكبير حجمهما على المحور x إلى 6 وتصغير الحجم على المحورين y و z إلى 0.5

4. بقي الآن الصناديق الثلاثة في منتصف الأرضية، وهي عبارة عن مكعبات بالحجم الافتراضي اثنان منها على الأرضية (y = 0.5) والثالث موضوع فوقهما (y = 1.5). بعد تحديد المواقع عموديا يمكن استخدام أدوات التحريك والتدوير من المشهد للوصول لنتيجة مشابهة للتي في الصورة، مع مراعاة كون الدوران على المحور y فقط بزوايا تتراوح بين 20- إلى 45 درجة.



الشكل 2: مكوّن Transform

يمكن الاطلاع على النتيجة في المشهد scene1 في المشروع المرفق. لاحظ أن كل ما قمنا به لبناء هذا المشهد هو تغيير الخصائص الموجودة في المكوّن Transform لجميع الأشكال التي قمنا بإضافتها. لعلك لاحظت أيضا وجود مكوّنات أخرى غير Transform مثل Mesh Filter و Mesh Renderer

وغيرها. وجود هذه المكوّنات وانتماؤها لكائن بعينه يلخص طريقة تعامل محرك Unity مع بيانات المشهد؛ حيث يتكون المشهد الكامل من مجموعة من الكائنات، ويحتوي كل منها على عدد غير محدد من المكوّنات التي تتفاعل فيما بينها لتعطي الشكل والسلوك المطلوب لهذا الكائن. خذ على سبيل المثال المكوّن Transform الذي تعاملنا معه آنفا، والذي اتضح لدينا أنه المسؤول عن تحديد موقع وحجم ودوران الكائن في الفضاء ثلاثي الأبعاد، بالإضافة إلى وظائف أخرى سنأتي على ذكرها لاحقا إن شاء الله. مثال آخر هو المكوّن Mesh Renderer والذي سنناقشه في الفصل الثالث من هذه الوحدة، وهو المسؤول عن تصيير الكائن في المشهد ويختص بكل ما يتعلق بعملية التصيير من معالجات. جرّب تعطيل هذا المكوّن في أحد الأشكال الموجودة في المشهد، ما النتيجة التي ستحصل عليها؟

لتعطيل أي مكوّن في أي كائن من كائنات المشهد، قم ببساطة بإزالة التحديد عن مربع الاختيار في أعلى يسار المكوّن في نافذة الخصائص

إذا كنت لا تزال ملتبسا حول بناء المشهد، والفرق بين الكائن والمكوّن وما الذي تراه في نافذة الخصائص وهرمية المشهد، لاحظ الشكل 3 والذي يلخص بناء المشهد في محرك Unity.



الشكل 3: العلاقة بين المشهد والكائنات والمكوّنات في محرك Unity

الفصل الثاني: الروابط بين الكائنات في المشهد

لعلك لاحظت خلال عملك على بناء الشكل السابق أنه يمكنك التعامل مع كل كائن في المشهد على حدة، دون أن يؤثر تغيير خصائص أي منها على الآخر. لكن لنفترض أنك تريد نقل هذا المشهد من منتصف الفضاء حيث قمت ببنائه إلى موقع آخر، أو أنك تريد عمل عدة نسخ منه في أماكن مختلفة. في الحالة الأولى يمكنك أن تقوم بتحديد كافة العناصر سواء من نافذة المشهد أو هرمية المشهد ونقلها دفعة واحدة، أمّا في الحالة الثانية يمكنك تحديد كافة كائنات المشهد ثم نسخها ولصقها.

الحل الأفضل هو دون شك بناء روابط منطقية بين هذه الكائنات بحيث يمكن التعامل معها أو مع بعضها كوحدة واحدة. في الفضاء ثلاثي الأبعاد، وفي معظم برامج التصميم ومحركات الألعاب، ترتبط الكائنات ببعضها البعض على شكل علاقة هرمية تتكون من آباء وأبناء، بحيث يتأثر الابن بخصائص الأب وليس العكس.

بالرجوع إلى الشكل 1، يمكننا ملاحظة وجود ثلاث أجزاء رئيسية: الأرضية، القوسان، الصناديق. لنأخذ الأرضية على أنها الجذر الذي ستتصل به كل الأجزاء الأخرى، كما علينا أن لا نغفل ربط الأعمدة بالعوارض بحيث يمكننا أن نعاملها كوحدة واحدة. لبناء هذه العلاقات يمكنك ببساطة سحب أي كائن من هرمية المشهد إلى داخل كائن آخر، بحيث يصبح "ابنا" له يتأثر بكل ما ينطبق على الأب من خصائص. لنقم بذلك بخطوات بسيطة:

1. قم بإضافة كائنين فارغين لنجمع في كل منهما أجزاء قوس من القوسين الموجودين في المشهد. يمكنك ذلك عن طريق قائمة Game Object > Add Empty.

2. قم بتغيير أسماء الكائنين الجديدين إلى Arc1 و Arc2 وذلك عن طريق نافذة الخصائص، وتأكد أيضا أنهما موجودان في الموقع (0, 0, 0).

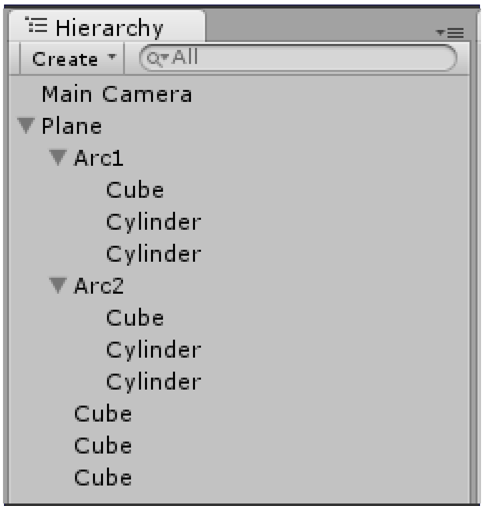
3. قم بسحب كل من الأسطوانتين والمكعب التي تشكل القوس الأول إلى داخل Arc1 وكذلك الكائنات التي تشكل القوس الثاني إلى داخل Arc2 وذلك عن طريق هرمية المشهد.

4. الآن قم بسحب كل من Arc1 و Arc2 إلى داخل كائن الأرضية المسمى Plane.

5. قم أخيرا بسحب المكعبات الثلاث (الصناديق) إلى داخل كائن الأرضية.

لعلك لاحظت أثناء العمل تشابه الأسماء بين كثير من الكائنات الموجودة في المشهد، لذا من الأفضل دائما تغيير أسمائها الافتراضية إلى أسماء أخرى بحيث يسهل تمييزها والوصول إليها، خاصة عندما يكبر المشهد ويصبح أكثر تعقيدا (سأترك هذه المهمة لك).

بعد الانتهاء من هذه التعديلات البسيطة، ستلاحظ أن هرمية المشهد أصبحت متفرعة كما في الشكل 4. وأن أي تعديل على الأب ينتقل تلقائيا للأبناء كالتحريك والتدوير، كما إن حذف الكائن الأب من المشهد يتبعه تلقائيا حذف الأبناء. لكن العكس غير صحيح، حيث أن أي تعديل على الكائن الابن لا ينعكس على الأب.



الشكل 4: الروابط بين الكائنات في المشهد كما تظهر في الهرمية

لو قمت بنقل كائن الأرضية من مكان لآخر داخل الفضاء، ستلاحظ أن قيم الموقع الخاصة في نافذة الخصائص تتغير، بينما لا تتغير القيم الخاصة بأي من الكائنات المضافة إليه كأبناء. السبب في ذلك هو أن خصائص الأبناء من موقع ودوران وقياس تكون نسبية إلى خصائص الأب، وتنتمي للفضاء المحلي للأب object space وليس الفضاء العالمي الذي يمثل المشهد كاملا world space. سنأتي لاحقا على الفروق بين هذين الفضاءين وأهميتها فلا تشغل بالك بها في الوقت الراهن.

الفصل الثالث: خصائص التصيير

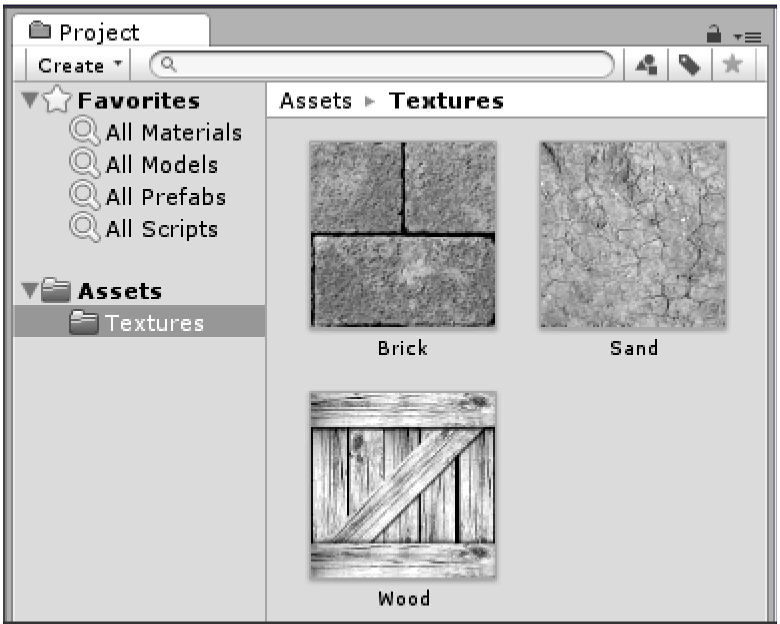
سنتحدث في هذا الفصل عن الخصائص الأساسية لتصيير الكائنات في محرك Unity. كما ذكرت سابقا فإن المكوّن المسؤول عن عملية التصيير هو Mesh Renderer، والذي ستجده فقط في الكائنات التي ينبغي أن تكون مرئية للاعب أثناء تشغيل اللعبة. فلن تجده مثلا في كائن الكاميرا (لأنها نفسها ليست مرئية)، كما لن تجده أيضا في الكائنين الفارغين Arc1 و Arc2 الذين قمنا بإضافتهما لأسباب تنظيمية تتعلق بالروابط بين الكائنات فقط. جدير بالذكر أن Mesh Renderer ليس المكوّن الوحيد المتعلق بالتصيير، بل توجد مكونات أخرى سنأتي على ذكرها في وقتها. سنكتفي حاليا بالحديث عن هذا المكوّن ونتعرف على أهم خصائصه.

أول العناصر التي سنتناولها في موضوع التصيير هي الإكساءات، وهي عبارة عن صور ثنائية الأبعاد يتم طلاؤها على المجسمات ثلاثية الأبعاد لتعطي المظهر المطلوب. على سبيل المثال، يمكن إن نكسو الأرضية بصورة تراب، والأعمدة والعوارض بصورة حجارة، والصناديق التي تتوسط المشهد بصورة سطح خشبي.الشكل 5 يوضح أمثلة على الإكساءات التي سنستخدمها في المشهد الذي قمنا ببنائه.

لإضافة الإكساءات إلى الكائنات الموجودة في المشهد يجب أولا إضافة الملفات التي تحتوي على هذه الإكساءات إلى المشروع الحالي، ومن ثم ربطها بالكائنات عن طريق المواد ومكوّن Mesh Renderer.

لإضافة ملفات جديدة إلى المشروع قم بسحب الملف من موقعه الحالي إلى داخل مستعرض المشروع Project كما في .الشكل 5 من الأفضل أن تخصص مجلدا فرعيا داخل المشروع خاصا بالإكساء (Textures مثلا) وذلك عن طريق النقر بزر الفأرة الأيمن على المجلد الجذري Assets ثم اختيار Create > Folder. لإضافة هذه الإكساءات إلى كائنات المشهد، قم ببساطة بسحب كل منها إلى الكائن المطلوب في نافذة المشهد

عند إضافة الإكساء إلى كائن معين يقوم Unity تلقائيا بإضافة مادة جديدة (Material) إلى المشروع وربط الإكساء بهذه هذه المادّة، ومن ثم إضافة المادة لمكوّن التصيير Mesh Renderer الخاص بهذا الكائن. جدير بالذكر أن الإكساء يرتبط بالمادة عن طريق المظلل Shader والذي يحدد كيف ستظهر المادة بشكلها النهائي. لتوضيح العلاقة بين هذه العناصر انظر الشكل 6 والذي يوضّح أن الكائن يحتوي على مكوّن Mesh Renderer والذي بدوره يحتوي على مادة أو أكثر. كل مادة من هذه المواد ترتبط بمظلل واحد. وهذا المظلل يحتوي على عدة خصائص من أهمها الإكساء، الذي يعطي التأثير الأكبر على الشكل النهائي. افتراضيا، يستخدم Unity مظلل التشتيت Diffuse والذي يقوم ببساطة بطلاء سطح الكائن بالإكساء دون أي مؤثرات أخرى، باستثناء إمكانية مزج لون الإكساء الأصلي مع لون آخر.

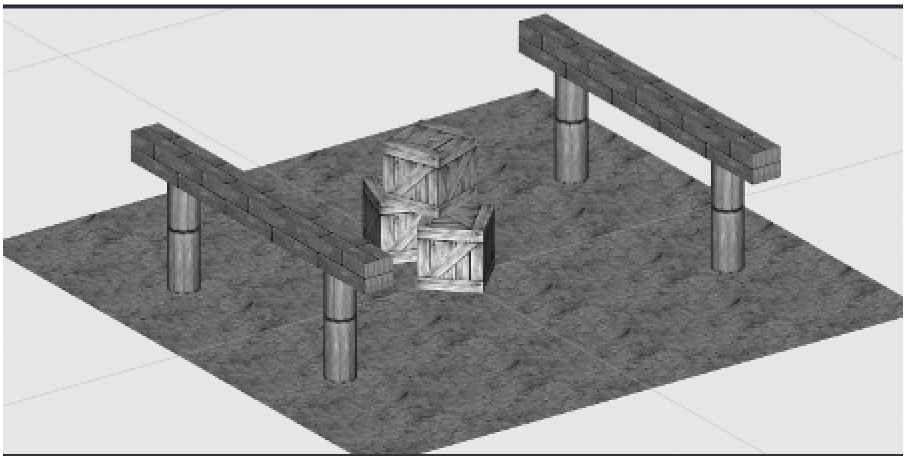


الشكل 5: إضافة ملفات الإكساء إلى المشروع



الشكل 6: العناصر المكونة لآلية تصيير الكائن

لعلك لاحظت بعد إضافة الإكساء إلى الكائنات أن Unity قد قام بإنشاء مجلد جديد اسمه Materials ليقوم بتخزين المواد فيه. عند اختيارك لأي مادة من هذه المواد تظهر خصائصها في نافذة الخصائص ومن ضمن هذه الخصائص قيم تكرار الإكساء (tiling) والتي تحدد كم عدد المرات التي يجب أن تتكرر فيها صورة الإكساء على المحورين الأفقي والعمودي، وما هي الإزاحة المطلوبة عليهما إن وجدت. يقوم Unity افتراضيا بتكرار الصورة مرة واحدة على كل سطح، ويمكن تحسين المظهر بزيادة عدد مرات التكرار لتصبح بالنسبة لمادة الأرضية 5 على كلا المحورين وبالنسبة لمادة الأعمدة والعوارض 5 على المحور الأفقي فقط. بالنسبة للصناديق لا ينبغي تغيير القيم لأن كل وجه يجب أن يعكس سطح صندوق واحد فقط. الشكل 7 يظهر المشهد النهائي بعد إضافة الإكساء والمواد ويمكن الاطلاع عليه في المشهد "scene1 textured" في المشروع المرفق.



الشكل 7: المشهد السابق بعد إضافة الإكساء والمواد وتعديل قيم التكرار

الفصل الرابع: أنواع الضوء وخصائصه

تعتبر الإضاءة من العناصر الأساسية التي تساهم في بناء المشهد. فمن خلالها يمكن تمييز المناطق المضاءة من المعتمة، وتكوين ظلال الأجسام، وغيرها من الاستخدامات مثل شد انتباه اللاعب لمكان معين في المشهد أو حتى استخدامها في عمل الألغاز.

ما يهمنا في هذا الفصل هو التعرف على أنواع الضوء التي يوفرها محرك Unity وكيفية الاستفادة منها في بناء المشهد. أول أنواع الضوء الذي سنتناوله هو الضوء المحيط (ambient light)، والذي يمثل لون الضوء الافتراضي على مستوى المشهد كاملا، دون إضافة أي مصادر ضوئية أخرى. بالتالي يمكن استخدامه لتمثيل الوقت في اليوم من نهار وليل أو شروق وغروب، وذلك عن طريق تغيير لون الإنارة.

نظرا لكون الضوء المحيط ذو تأثير على مستوى المشهد ككل، فإنه لا يرتبط بكائن معين في المشهد،

بل يمكن تغيير قيمته من نافذة إعدادات التصيير (Render Settings) والتي تحتوي على عناصر ذات تأثير على مستوى المشهد ككل.

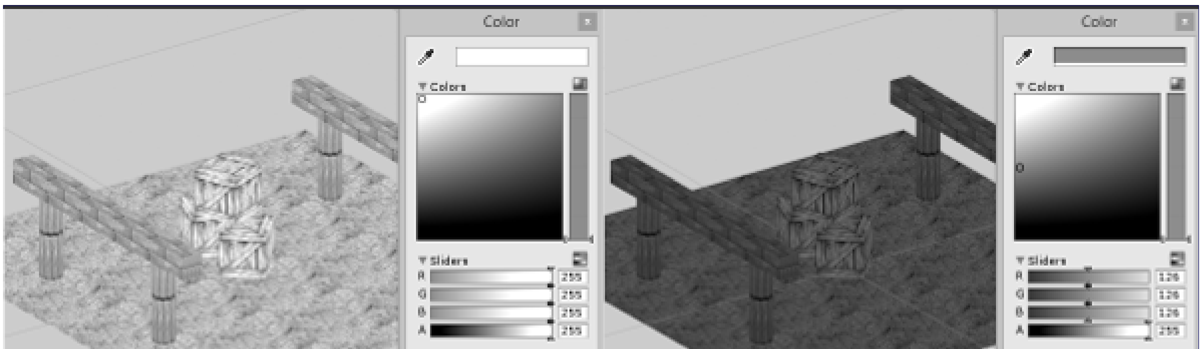
لتغيير قيمة الضوء المحيط قم باستدعاء نافذة إعدادات التصيير عن طريق القائمة Edit > Render Settings حيث ستظهر الإعدادات داخل نافذة الخصائص. يمكنك بعد ذلك تغيير قيمة اللون في العنصر Ambient Light

لمشاهدة أي تغيير تقوم به على الضوء المحيط أو أي أمور أخرى تتعلق بالإضاءة، لا بد أولا من تشغيل الإضاءة في المشهد. أقصد بتشغيل الإضاءة هنا أن تطلب من Unity أن يأخذ بعين الاعتبار المؤثرات الضوئية عند تصييره للمشهد. عدا ذلك فإن هذه المؤثرات ستُهمل وسيبدو المشهد كله على درجة واحدة ولون واحد من الإضاءة.

يمكن إيقاف وتشغيل الإضاءة في المشهد عن طريق الزر والموجود في أعلى نافذة المشهد.

قد يبدو مربكا للوهلة الأولى إعتام المشهد عند تشغيل الإضاءة؛ ذلك أن الضوء المحيط معتم افتراضيا. جرب تغيير لون الإنارة الخاصة به إلى قيمة قريبة من الأبيض لتلاحظ أن مستوى الإضاءة في المشهد يرتفع. الشكل 8 يوضح كيف يتأثر المشهد بتغيير لون الإضاءة المحيطة.

من الناحية الفنية، تكمن أهمية لون الضوء المحيط في المشهد في الشعور العام الذي يعطيه للاعب، فمثلا اللون الأحمر يدل على حرارة عالية ويستخدم في البيئات البركانية، والأزرق يدل عل البرودة والرعب، والأخضر يستخدم في بيئات كالمستنقعات أو الأماكن الرطبة ليعطي الشعور المناسب. في حالة مشهدنا الحالي سنترك هذا الضوء على قيمته الافتراضية لأننا سنضيف مصادر ضوئية أخرى للمشهد.



الشكل 8: تأثير لون الضوء المحيط على المشهد

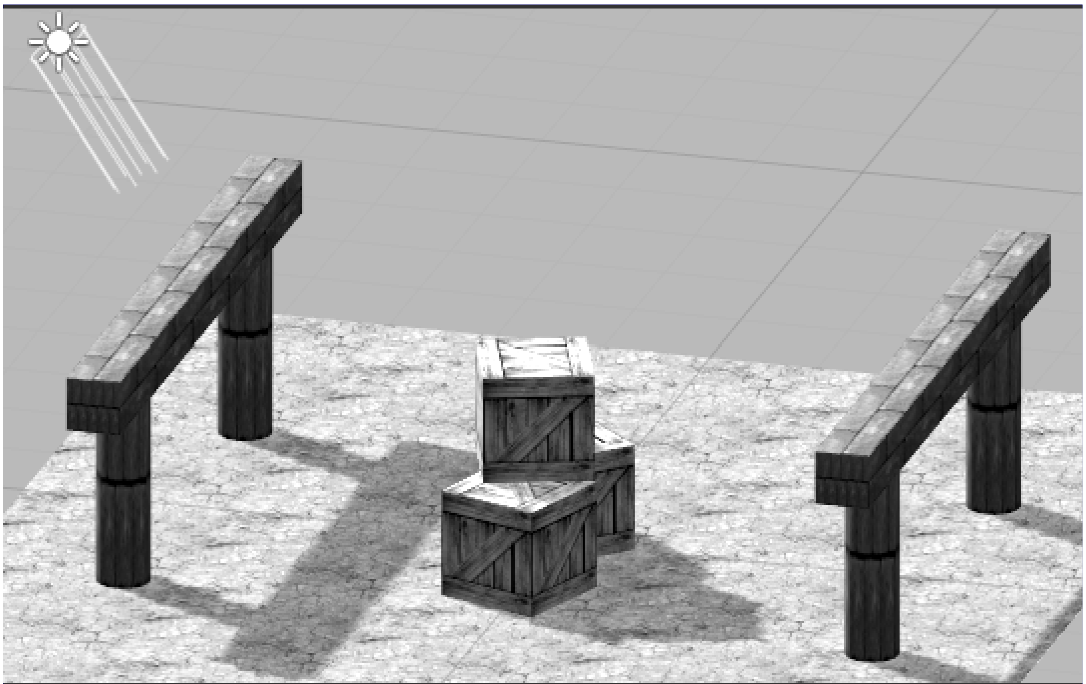
النوع الثاني الذي سنتناوله هو الضوء الاتجاهيّ (Directional Light)، والذي ينبغي استخدامه مرة واحدة في كل مشهد ليمثل المصدر الرئيسي والأقوى للإضاءة، مثل الشمس في المشاهد النهارية

والبدر في المشاهد الليلية. يتكون الضوء الاتجاهي من خصائص إضافية تختلف عن تلك في الضوء المحيط، حيث يمكن تغيير شدة الإنارة وزاويتها بالإضافة للون.

لنقم الآن بإضافة ضوء اتجاهي للمشهد الذي نعمل عليه، ونحاول تغيير بعض القيم الخاصة بالإنارة.

لإضافة ضوء اتجاهي للمشهد. ادخل إلى القائمة Game Object > Create Other > Directional Light ثم قم باختياره من الهرمية للتعديل على القيم في نافذة الخصائص

من الجدير بالذكر أن الضوء الاتجاهي يؤثر على المشهد كله بصورة متساوية ولا ينحصر تأثيره بمكان معين، لذا فإن تغيير موقعه في فضاء المشهد لن يغير شيئا. كذلك الأمر بالنسبة للحجم. بالنسبة للاستدارة فالأمر مختلف، حيث تمثل استدارة كائن الضوء الاتجاهي الزاوية التي يسقط بها شعاعه على المشهد. قم مثلا بضبط قيمة الاستدارة الخاصة بكائن الضوء على (50, -45, 0) لجعل ميلان شعاع الضوء واضحا. يسهل عليك Unity معرفة الاتجاه الحالي للإشعاع الضوئي عن طريق رسم مجموعة خطوط باتجاه الإشعاع كما في الشكل 9.

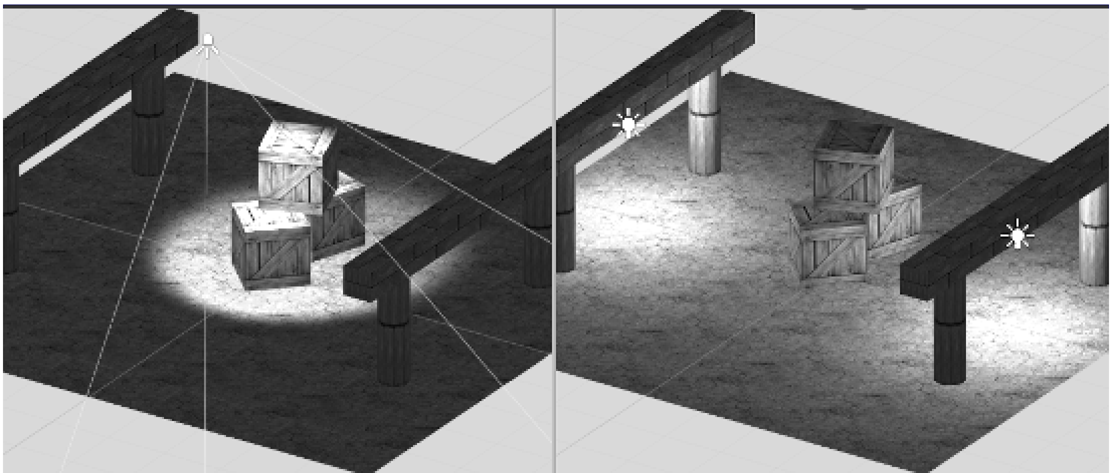


الشكل 9: المشهد بالمؤثرات الضوئية

من الخصائص الأخرى المهمة في الضوء الاتجاهي كثافة الإضاءة (Intensity) وهي تؤثر على قوة تأثير الضوء على الأجسام، لنضعه على قيمة منخفضة نسبيا: 0.6 على سبيل المثال؛ وذلك ليكون التأثير معقولا. يمكن أيضا تغيير اللون كما في حال الضوء المحيط لكننا لن نقوم بذلك الآن. يمكن أيضا تحديد

نوع الظلال من خلال Shadow Type وتعيين شدة قتامتها من خلال Strength. جرب مثلا تحديد Hard Shadows وضبط شدتها على 0.5. الشكل 9 يوضح المشهد النهائي بعد إضافة الضوء الاتجاهي.

النوعان الآخران من الإضاءة هما الضوء النقطي Point Light والضوء البقعيّ Spot Light. على الرغم من أننا لن نستعملهما في المشهد الحالي، إلا أنه لا بأس من التعرف عليهما قبل إنهاء هذا الفصل المتعلّق بالإضاءة. الشكل 10 يعرض مثالا لكل من هذين النوعين من الضوء. لاحظ أن الضوء النقطي يشع بشكل متساو في كل الاتجاهات تماما كالمصباح الكهربائي، والضوء البقعيّ يشع باتجاه واحد بحزم متباعدة أشبه بالأنوار الكاشفة وأضواء السيارات. بإمكانك استكشاف خصائصهما بنفسك.

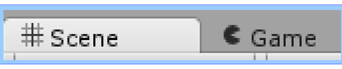


الشكل 10: لضوء النقطي (يمين) والضوء البقعي (يسار) وتأثيرهما على إنارة المشهد

الفصل الخامس: الكاميرا

بعد الانتهاء من بناء المشهد، من الضروري معرفة كيف سيبدو للاعب عند تشغيل اللعبة. منذ البداية ونحن نتعامل مع المشهد من خلال النافذة الخاصة بالمحرر Scene بينما هناك نافذة أخرى تُظهر لنا كيف سيرى اللاعب المشهد عند تشغيل اللعبة، ألا وهي نافذة اللعبة Game. يمكنك التبديل بينهما في أي وقت لترى الفرق، علما بأن Unity يقوم بالانتقال بينهما تلقائيا عندي تشغيل وإيقاف تشغيل اللعبة.

للتبديل بين نافذة المشهد ونافذة اللعب استخدم الألسنة في أعلى نافذة المشهد



الفرق بين النافذتين أن نافذة المشهد تعطيك الحرية في الإبحار في الفضاء والنظر للمشهد من جميع الزوايا، بينما لا تعرض لك نافذة اللعب سوى ما يراه اللاعب من خلال الكاميرا. باختيار كائن الكاميرا من المشهد، يمكننا الاطلاع على خصائصها في شاشة الخصائص، والتي سأشرح أهمها فيما يأتي:

1. الخلفية Background: وهو اللون الذي يظهر في أفق المشهد، أي في المناطق الفارغة التي لا يوجد بها أي كائنات مرئية للكاميرا.

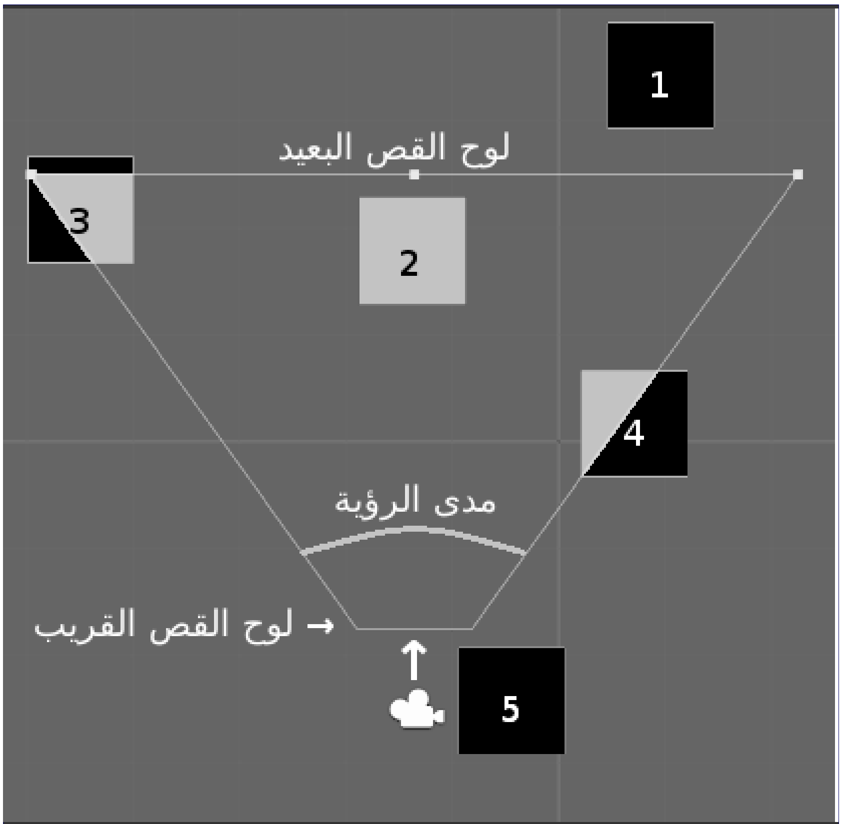
2. الإسقاط Projection: وهي الخاصية التي تحدد إذا ما كانت المسافة بين الكائن والكاميرا تؤثر على كيفية تصييره. ففي حالة الإسقاط المنظوري Perspective Projection تبدوا الأشكال البعيدة أصغر من القريبة مما يجعله الرؤية أكثر واقعية وشبها بنظر الإنسان. أما الإسقاط العمودي Orthogonal Projection فيقوم بتصيير كافة الأشكال بحجمها الأصلي بغض النظر عن بعدها أو قربها من موقع الكاميرا، ولهذا السلوك فوائد عديدة سنأتي على ذكرها، خاصة عند تطوير ألعاب ثنائية الأبعاد.

3. مجال الرؤية Field of View.

4. لوحا القص القريب والبعيد Near/Far Clipping Plane.

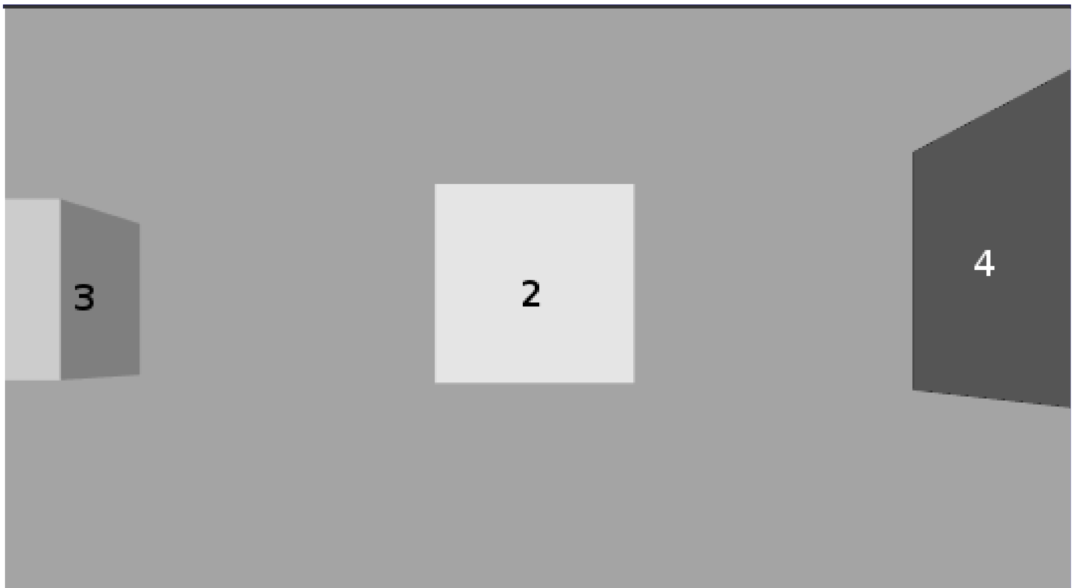
يرتبط مجال الرؤية بالإسقاط المنظوري، وهو الافتراضي في Unity. في حالة الإسقاط المنظوري يكون مجال رؤية الكاميرا على شكل هرم ناقص رأسه في مركز الكاميرا. وقاعدته هي لوح القص البعيد. يسمى هرما ناقصا لأن جزءا من أعلى رأس الهرم يكون مقطوعا بفعل لوح القص القريب. أما مجال الرؤية فيمثل قياس الزاوية بين الجانبين الأيمن والأيسر للهرم. الشكل 11 يوضح أجزاء الهرم وكيف يقوم بتحديد الكائنات المرئية وغير المرئية. حيث لا ترى الكاميرا سوى ما يقع داخل حدود هذا الهرم.

عند تشغيل اللعبة فإن اللاعب لن يرى في كل لحظة إلا ما يمكن للكاميرا رؤيته، لذا يجب دائما مراعاة تحريك الكاميرا بشكل مناسب أثناء اللعب وهو ما سنتعلمه إن شاء الله في هذا الكتاب.



الشكل 11: هرم الرؤية الخاص بالكاميرا، المناطق السوداء تمثل الجزء غير المرئي من الكائنات

الشكل 12 يوضح المشهد كما تراه الكاميرا الموجودة في الشكل 11، لاحظ أن أجزاء من المربعات ذات الأرقام 2 و 3 و 4 هي المرئية فقط.



الشكل 12: المشهد في الشكل 11 كما تراه الكاميرا

الفصل السادس: التحكم بخصائص الكائنات

بعد أن قمنا ببناء المشهد وتحديد إحداثيات الكائنات الموجودة فيه، يمكننا الآن الدخول قليلا إلى البرمجة لنرى كيف يتم تغيير قيم هذه الإحداثيات للحصول على التأثيرات المرغوبة. لنبدأ مثلا بتغيير مواقع الأشياء ولتكن بدايتنا مع الكاميرا التي تقوم بتصيير المشهد للاعب. الخطوة الأولى ستكون إضافة بُرَيمِج (script) إلى المشروع، ومن الأفضل إنشاء مجلد خاص بالبريمجات باسم Scripts تماما كما فعلنا مع الإكساءات بإضافة المجلد Textures. بعد إضافة المجلد المذكور يمكننا أن نبدأ بإضافة البريمجات المطلوب إلى داخله، ولنبدأ مع البريمج الأول CameraMover

> لإضافة بريمج جديد قم ببساطة بالنقر بزر الفأرة الأيمن على المجلد المطلوب (Scripts) ومن ثم Create > C# Script ثم قم بتسمية الملف باسم CameraMover

أود لفت الانتباه في هذه المرحلة إلى أن محرك Unity يدعم 3 لغات برمجة مختلفة، لكنني في هذا الكتاب سأتعامل مع لغة C# فقط. إن كنت مهتما بالكتابة بلغة أخرى يمكنك إعادة كتابة المنطق البرمجي بما يتلاءم مع اللغة التي تريدها. جدير بالذكر أيضا أنني أنصح باستخدام برنامج MonoDevelop لكتابة الأوامر البرمجية، وهو المحرر المضمّن مع Unity، بدلا من استخدام Visual Studio.

السرد 1 يمثل القالب الافتراضي للبريمج والذي سيقوم Unity بإنشائه عند إضافة البريمج الجديد للمشروع.

```
using UnityEngine;
using System.Collections;

public class CameraMover : MonoBehaviour {

	// Use this for initialization
	void Start () {}

	// Update is called once per frame
	void Update () {}
}
```

السرد 1: القالب الافتراضي للبريمجات في Unity

يختصر Unity علينا الطريق بإضافة الدّالتين الأكثر استخداما وهما Start() و Update()، حيث تستدعى الأولى مرة واحدة عند بداية التشغيل، لذا تستخدم لتعيين القيم الأولية قبل الدخول في دورة تحديث اللقطات، بينما تستدعى الثانية في كل مرة يتم فيها تصيير لقطة جديدة لتقوم بتنفيذ التغييرات المطلوبة على الكائن الذي تتبعه.

من المهم إدراك حقيقة أن البريمجات في Unity هي عبارة عن مكوّنات يتم التعامل معها داخل المحرك كما يتم التعامل المكوّنات الأخرى مثل Transform و Mesh Renderer التي تعرّفنا عليها في فصل سابق. من المهم أيضا ملاحظة أن كل بريمج في Unity يجب أن يكون وارثا من MonoBehavior حتى يتم التعرّف عليه كمكوّن ومعاملته على هذا الأساس. إذا لم تكن صاحب خبرة في البرمجة ولست متأكدا من دلالة مصطلح الوراثة هنا فلا بأس، يكفي أن نقول ببساطة أنك ينبغي أن تحافظ على البنية التركيبية للقالب أعلاه عند كتابة بريمجاتك حتى تعمل بالشكل الصحيح.

الخطوة التالية هي إضافة البريمج الذي أنشأناه كمكوّن على كائن الكاميرا، وبذلك فإن كل التغييرات التي سيقوم بها ستنفذ على هذا الكائن بالتحديد.

لإضافة بريمج إلى كائن محدد، قم باختيار الكائن المطلوب من هرمية المشهد ثم قم بسحب ملف البريمج من مستعرض المشروع إلى داخل نافذة خصائص الكائن. يمكن أيضا استخدام الزر Add Component أسفل نافذة الخصائص ومن ثم كتابة اسم ملف البريمج

بعد إضافة البريمج لكائن الكاميرا أصبحنا جاهزين لكتابة الأوامر البرمجية التي تحدد السلوك المطلوب. لنفرض أننا نريد أن تبدأ الكاميرا بالارتفاع لأعلى عند بداية تشغيل اللعبة. لتحقيق هذا الغرض علينا أن نحرك الكاميرا على المحور y لمسافة معينة عند تحديث كل لقطة. يمكننا أيضا أن نضيف متغيرا للتحكم بسرعة تحرك الكاميرا. السرد 2 يوضح الأوامر الضرورية لتحريك الكاميرا لأعلى. قم بفتح ملف البريمج على محرر MonoDevelop وذلك بالنقر المزدوج عليه في مستعرض المشروع ثم أجر التعديلات التي تراها في السرد 2.

من المهم هنا الحديث – ولو بشكل مختصر - عن الآلية التي يتم بها تشغيل اللعبة عن طريق محرك Unity. عند بداية التشغيل يقوم Unity بتهيئة البريمجات الفعّالة في المشهد وذلك عن طريق استدعاء الدّالّة ()Start المسؤولة عن تنفيذ الخطوات الأولية التي تؤدي لتشغيل البريمج بشكل صحيح وتنفيذ الوظيفة المطلوبة منه (مثل إعطاء القيم الأولية للمتغيرات). يتم استدعاء هذه الوظيفة مرة واحدة فقط عندما يتم تشغيل البريمج للمرة الأولى. بعد ذلك تبدأ حلقة تكرارية من بناء وتصيير اللقطات. يتم في كل دورة من دورات هذه الحلقة تنفيذ منطق اللعبة: ابتداء من قراءة مدخلات اللاعب، وتشغيل الذكاء الاصطناعي، والتحريك، وغيرها، حتى ينتهي بتصيير اللقطة على الشاشة ومن ثم الدخول في دورة تصيير جديدة. تتكرر هذه العملية بمعدل يجب أن لا يقل عن 25 إلى 30 مرة في الثانية حتى تظهر الحركة بشكل انسيابي للاعب. يقوم Unity في كل دورة باستدعاء الدّالّة ()Update مرة واحدة من كل البريمجات الفعّالة في المشهد، ويستمر الاستدعاء حتى يتم إيقاف تشغيل اللعبة.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class CameraMover : MonoBehaviour {
5.
6.       public float speed = 1;
7.
8.       // تستدعى هذه الدّالّة مرة واحدة عند بداية التشغيل
9.       void Start () {
10.
11.      }
12.
13.      // تستدعى هذه الدّالّة مرة واحدة في كل لقطة
14.      void Update () {
15.            transform.Translate(0, speed * Time.deltaTime, 0);
16.      }
17. }
```

السرد 2: تحريك الكائن لأعلى بسرعة ثابتة

في السطر 6 قمنا بتعريف متغير (أي وحدة برمجية لتخزين البيانات) من نوع float - أي عدد كسري – وأعطينا هذا المتغير القيمة 1. بعد ذلك استخدمنا المتغير في السطر 15 لتحريك الكائن باستدعاء الدّالّة ()transform.Translate والتي وظيفتها إزاحة الكائن في الفضاء على المحاور الثلاث x, y, z تبعا للقيم التي يتم تزويدها عند الاستدعاء. لاحظ أننا حددنا قيمة للإزاحة على المحور العمودي y فقط.

في السطر 15 نحتاج لأن نقوم بتحريك الكائن نحو الأعلى بمسافة محددة في كل إطار، وعلينا أن نحسب مقدار هذه المسافة. جميعنا نعرف القانون الفيزيائي الذي ينص على أن السرعة تساوي المسافة المقطوعة في زمن معين، بالتالي لكي نحسب المسافة علينا أن نضرب السرعة (وهي قيمة المتغير speed) بالزمن المنقضي. لكن ما هو هذا الزمن؟ بما أن الإزاحة ستتكرر عند تصيير كل إطار، فإن الزمن هنا هو الزمن المنقضي منذ أن تم تصيير الإطار الماضي، والذي نحصل عليه عبر المتغير Time.deltaTime. فعندما نضرب هذا الزمن المنقضي بالسرعة نحصل على المسافة المطلوبة للإزاحة، وهذا تحديدا ما نفعله. قم الآن بتشغيل اللعبة لترى تأثير هذا البريمج على حركة الكاميرا في المشهد.

لتشغيل اللعبة أو إيقاف تشغيلها استخدم الزر 

الشكل 13 يوضح كيف يظهر مكوّن البريمج في نافذة الخصائص وكيف أن المتغيرات العامّة (أي تلك التي يتم تعريفها باستخدام كلمة public) تظهر على شكل خانات يمكن تغيير قيمها الأولية مباشرة دون الحاجة لتغيير القيمة داخل الكود في كل مرة. يمكنك الآن تغيير سرعة حركة الكاميرا بسهولة عن طريق تغيير القيمة في الخانة Speed. جرب أن تغير السرعة إلى قيمة سالبة وشاهد النتيجة التي تتوقعها.



الشكل 13: مكوّن البريمج كما يظهر في نافذة الخصائص

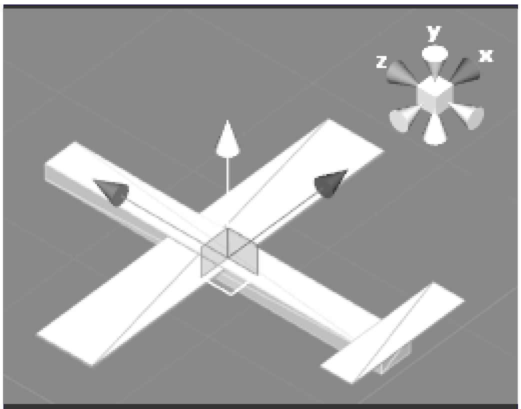
عند تشغيل اللعبة ستلاحظ أن الكاميرا تبدأ مباشرة في الارتفاع حسب السرعة المحددة مما سيؤدي لاختفاء المشهد عن ناظر اللاعب بعد فترة بسيطة. لنحاول تركيز نظر الكاميرا على وسط المشهد، وذلك عن طريق تغيير استدارتها بعد التحريك لتنظر إلى نقطة الأصل. لحسن الحظ يريحنا Unity من عناء حساب الاستدارة المطلوبة بتوفير الدّالّة ()transform.LookAt والتي نقوم باستدعائها وإعطائها نقطة معينة في الفضاء لتنظر إليها. قم بإضافة السطر التالي بعد السطر 15 في السرد 2

```
transform.LookAt(Vector3.zero);
```

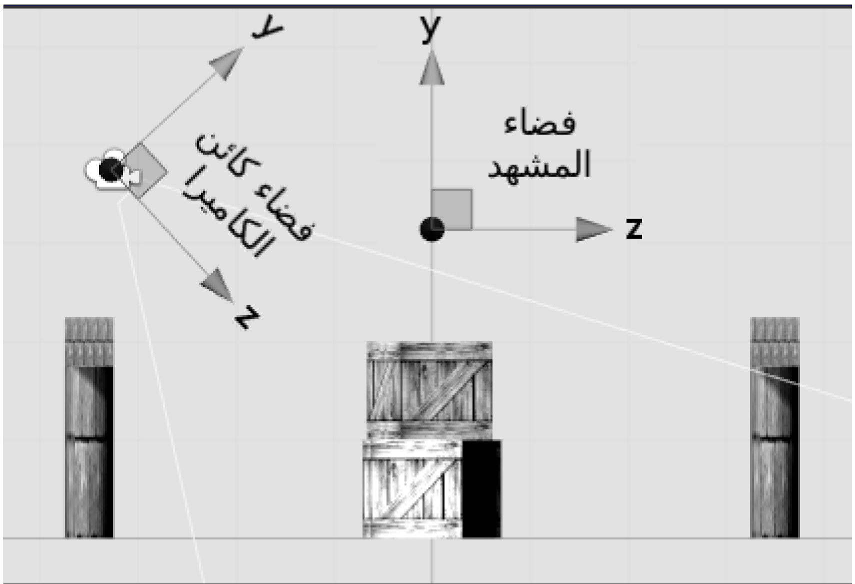
ستلاحظ أن الكاميرا تبقى مركزة على وسط المشهد وهو نقطة الأصل بينما تواصل ارتفاعها. لعلك لاحظت أيضا أنها تقترب شيئا فشيئا من وسط المشهد على المحورين x و z إلى أن تصبح فوقه تماما، عندها يحدث اضطراب في الصورة بسبب تغير استدارة الكاميرا بسرعة.

السبب في السلوك أعلاه هو كون الإزاحة التي تقوم بها دالّة ()transform.Translate تخضع لفضاء الكائن المحلي وليس فضاء المشهد أو الفضاء العالمي. فضاء الكائن أو الفضاء المحلي يُقصد به محاور الكائن نفسه، فلو تخيلت الكائن طائرة بجناحين كالتي في الشكل 14، فإن مقدمة الطائرة تشير للاتجاه الموجب للمحور z، وجناحها الأيمن يشير للاتجاه الموجب للمحور x، والاتجاه الموجب للمحور y يمتد من سطح الطائرة العلوي بشكل متعامد عليه نحو الأعلى، بالتالي كيفما تحركت هذه الطائرة أو مالت واستدارت فإن هذه المحاور تتحرك معها وتستدير. (وعلى سبيل التحفيز، فإننا سنقوم ببناء هذه الطائرة وجعلها تطير في الوحدة التالية إن شاء الله).

بالعودة لمثالنا السابق، عندما قمنا بإدارة الكاميرا لتنظر نحو الأسفل، فإن هذا الدوران جعل المحور العمودي y الخاص بالكاميرا مائلا عن المحور العمودي الخاص بفضاء المشهد. نتيجة لذلك، فإن الإزاحة تصبح مائلة شيئا فشيئا. الشكل 15 يوضح محاور فضاء كائن الكاميرا بعد دورانها نحو الأسفل.



الشكل 14: محاور الفضاء المحلي لنموذج طائرة



الشكل 15: الفرق بين فضاء المشهد والفضاء المحلي للكائن، لاحظ التغير في محاور فضاء الكاميرا بعد استدارتها نحو الأسفل لتنظر لمنتصف المشهد

لحل هذه المشكلة، علينا ببساطة أن نخبر الدّالة ()transform.Translate أن تستخدم فضاء المشهد للإزاحة بدلا من فضاء الكائن. لذا علينا أن نقوم بتغيير السطر 15 في السرد 2 ليصبح كما يلي.

```
transform.Translate(0, speed * Time.deltaTime, 0, Space.World);
```

بعد هذا التعديل ستتحرك الكاميرا للأعلى فقط بغض النظر عن دورانها، ذلك أن محاور فضاء المشهد ثابتة دائما. لنقم الآن بشيء أكثر إثارة من مجرد تحريك الكاميرا. ماذا لو غيرنا دوران الضوء الاتجاهيّ الذي يضيء المشهد؟ كيف سيؤثر ذلك على ظلال الأجسام؟ لعمل ذلك علينا أن نضيف بريمجا جديدا

للمشروع ليقوم بمهمة إدارة هذا الضوء، ولنسمه مثلا LightRotator. يمكنك مشاهدة هذا البريمج في السرد 3.

```
using UnityEngine;

public class LightRotator : MonoBehaviour {

	public float speed = 10;

	void Update () {
		transform.Rotate(speed * Time.deltaTime, 0, 0);
	}
}
```

السرد 3: البريمج الخاص بتدوير الضوء الاتجاهيّ

يقوم هذا البريمج ببساطة بتدوير الضوء حول محوره المحلي x، وبما أن اتجاه إشعاع الضوء هو في الاتجاه الموجب لمحوره المحلي z، فإن هذا التدوير سيؤدي لتغيير الزاوية التي يسلط بها الضوء على المشهد، فحسب قاعدة اليد اليسرى، أنت تدير الضوء حول إصبع الوسطى ويشير أصبع السبابة إلى اتجاه الإشعاع. هذا الدوران سيجعلك تلاحظ أن الضوء يصبح مائلا شيئا فشيئا ويزيد طول الظلال حتى يختفي الضوء تماما كما عند غروب الشمس، ثم يعود مجددا كما في الشروق. استعملت هنا سرعة عالية نسبيا (10 درجات في الثانية) حتى تلاحظ التغير في ميلان الضوء قبل أن تبتعد الكاميرا عن المشهد.

تحدثنا في هذه الوحدة عن كيفية بناء مشهد أساسي مؤلّف من كائنات متوزعة في الفضاء، وخصاص تصيير هذه الكائنات. كما تناولنا كيفية ربط هذه الكائنات ببعضها البعض وأنواع الضوء التي يمكننا استخدامها في بناء المشهد. كما تناولنا بشكل أولي كيفية كتابة بريمجات بلغة C# و استخدام هذه البريمجات لتغيير خصائص الكائنات أثناء اللعب.

لم نتطرق بطبيعة الحال لكل ما يتعلق بالمؤثرات البصرية المتاحة؛ ذلك أن الهدف من الوحدة كان التعرف على كيفية تركيب المشهد لغرض الدخول بأسرع وقت إلى التفاعل مع كائنات المشهد والذي يشكل أساسا في تطوير الألعاب. هناك الكثير من الأمور التي تتعلق بالتصيير والإكساء والمظللات التي سنتطرق إليها في الوحدات القادمة إن شاء الله.

تمارين

1. تحدثنا عن استخدام كائنات الأشكال الأساسية من أجل بناء مشهد بسيط. حاول أن تستخدم هذه الأشكال لبناء مشهد أكبر من الذي تعاملنا معه في هذه الوحدة. حاول أن ترسم حديقة بجدار مثلا مع منزل صغير في داخلها، يمكنك البحث في الإنترنت عن إكساءات مناسبة.

2. ناقشنا ببعض التفصيل نوعين من الضوء: الضوء المحيط والضوء الاتجاهي. أضف إلى المشهد في التمرين رقم 1 أضواء نقطية لإنارة المشهد. واجعل الضوء المحيط منخفضا لتجعل البيئة ليلية. حاول إضافة هذه الأضواء في أماكن مناسبة من المشهد واضبط قيم خصائصها بما يناسب ما تمثله في كل موقع.

3. قم بتغيير البريمج CameraMover بحيث يجعل الكاميرا تدور حول المشهد بشكل أفقي دون أن ترتفع أو تنخفض، مع إبقاء دورانها متوجها نحو نقطة الأصل بحيث تعرض المشهد للاعب بغض النظر عن موقعها. ستساعدك محاور الفضاء المحلي للكائن في هذه المهمة. بعد الانتهاء من التعديل قم بإضافة البريمج للكاميرا في المشهد الذي قمت بإنشائه في التمرين رقم 1.

4. حاول الاستفادة من العلاقات بين الكائنات، لربط الكاميرا في التمرين رقم 3 بضوء بقعي يتم تسليطه في الاتجاه الذي تنظر إليه الكاميرا بحيث يدور معها أينما دارت.

الوحدة الثانية: قراءة مدخلات اللاعب

تعلمنا في الوحدة السابقة الأمور الأساسية المتعلقة ببناء مشهد بسيط وكيفية التعامل مع خصائص الكائنات التي تشكل هذا المشهد. تعرفنا أيضا على الإكساءات وأنواع مختلفة من الإضاءة، وتعاملنا مع الكاميرا التي تشكل عين اللاعب على المشهد. وتعاملنا مع البريمجات وشاهدنا قدرتها على تغيير خصائص الكائنات أثناء اللعب.

في هذه الوحدة سنتطرق إن شاء الله إلى أول خطوة في عملية الوصول إلى بناء لعبة كاملة، وهي قراءة مدخلات اللاعب. فالألعاب وإن اختلفت أنواعها وميكانيكياتها، إلّا أنها من أبسطها إلى أكثرها تعقيدا لا تستغني عن قراءة مدخلات اللاعب بطريقة أو بأخرى؛ لأن التفاعل مع اللاعب أمر أساسي في أي لعبة. سنتعرف على طرق مختلفة لقراءة المدخلات، وسيكون معظم العمل مرتكزا على البريمجات التي ستقرأ المدخلات وتستجيب لها بالطريقة المناسبة داخل المشهد.

ما يتوقع منك عند الانتهاء من دراسة هذه الوحدة:

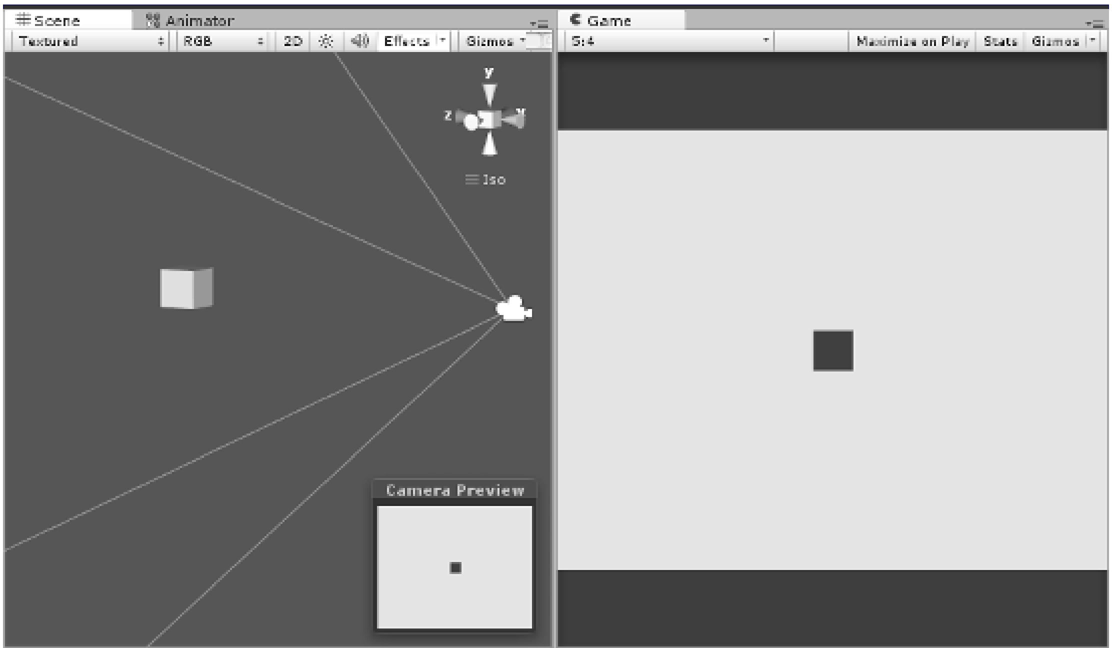
- قراءة المدخلات من لوحة المفاتيح
- تطبيق نظام إدخال ألعاب المنصات (platformer)
- قراءة المدخلات من الفأرة
- تطبيق النظر باستخدام الفأرة ونظام إدخال ألعاب منظور الشخص الأول (first person)
- تطبيق نظام إدخال ألعاب منظور الشخص الثالث (third person)
- تطبيق نظام تحكم ألعاب سباق السيارات
- تطبيق نظام تحكم ألعاب محاكاة الطيران

الفصل الأول: قراءة مدخلات لوحة المفاتيح

تعتبر لوحة المفاتيح أهم أجهزة الإدخال الخاصة بأجهزة الحاسوب الشخصي، فلا تكاد تخلوا لعبة من هذه الألعاب من الاعتماد على مجموعة من المفاتيح التي تقوم بوظائف أساسية. معظم ألعاب الشركات الكبيرة تسمح لك بتغيير إعدادات لوحة المفاتيح بغرض تغيير الارتباط بين المفتاح والوظيفة مما يعطي اللاعب مساحة جيدة لتخصيص اللعبة بما يناسبه.

يساعدنا Unity على قراءة مدخلات لوحة المفاتيح بطريقتين: الطريقة الأولى هي قراءة رمز المفتاح مباشرة، بأن يخبرك أن اللاعب يقوم حاليا بالضغط على المفتاح A أو Z مثلا. أما الطريقة الثانية، فهي أن تقوم كمطور بإنشاء وظائف وربطها مع مفاتيح، بحيث يمكنك لاحقا أن تسمح للاعب بتخصيص المدخلات بما يناسبه. سأتناول في هذا الفصل الطريقة الأولى فقط، وذلك انطلاقا من رؤية الكتاب بأن يكون عامّا قدر الإمكان وتجنب الدخول في الوظائف المخصصة بمحرك Unity، مما يجعله ذا فائدة للمطورين الذين يستعملون غيره من المحركات.

لتعلّم كيفية قراءة المدخلات، لنقم بإنشاء مشهد جديد في المشروع الذي بدأناه في الوحدة السابقة، أو إنشاء مشروع جديد كليا. ما نحتاجه مبدئيا هو مشهد يحتوي فقط على الكاميرا بموقعها الأصلي (0, -10, 0) وكائن مكعب في نقطة الأصل، مما يجعله مرئيا من قبل الكاميرا في منتصف الشاشة كما في الشكل 16.



الشكل 16: مشهد بسيط بهدف قراءة مدخلات لوحة المفاتيح

بعد الانتهاء من بناء المشهد قم بإنشاء بريمج جديد في المجلد الفرعي scripts والذي قمنا بتخصيصه للبريمجات، وليكن اسمه KeyboardMovement. لنفرض أننا نريد أن نتحكم بموقع المكعب عن طريق لوحة المفاتيح، وسنستخدم مفاتيح الأسهم الأربعة لتحريكه في الاتجاه المناسب، يمينا أو يسارا، أو أعلى أو أسفل. السرد 4 يوضح الكود المطلوب لتطبيق هذه الحركة. قم بإضافة البريمج إلى كائن المكعب في المشهد ومن ثم قم بتشغيل اللعبة لتجرب تحريكه باستخدام الأسهم.

```
using UnityEngine;
```

```
2. using System.Collections;
3.
4. public class KeyboardMovement : MonoBehaviour {
5.        //سرعة الحركة
6.        public float speed = 10;
7.
8.        // Use this for initialization
9.        void Start () {
10.
11.       }
12.
13.       // Update is called once per frame
14.       void Update () {
15.             //للأعلى
16.             if(Input.GetKey(KeyCode.UpArrow)){
17.                   transform.Translate(0, speed * Time.deltaTime, 0);
18.             }
19.             //للأسفل
20.             if(Input.GetKey(KeyCode.DownArrow)){
21.                   transform.Translate(0, -speed * Time.deltaTime, 0);
22.             }
23.             //لليمين
24.             if(Input.GetKey(KeyCode.RightArrow)){
25.                   transform.Translate(speed * Time.deltaTime, 0, 0);
26.             }
27.             //لليسار
28.             if(Input.GetKey (KeyCode.LeftArrow)){
29.                   transform.Translate(-speed * Time.deltaTime, 0, 0);
30.             }
31.       }
32. }
```

السرد 4: بريمج لقراءة مدخلات أسهم لوحة المفاتيح وتحويلها لحركة

بقراءة السرد 4 نلاحظ أن قراءة مدخلات اللاعب عملية مستمرة طالما أن اللعبة تعمل، لذا علينا أن نقوم بهذه العملية من داخل الدّالّة ()Update. نلاحظ أيضا في الأسطر 16، 20، 24، 28 أننا نستخدم الدّالّة () Input.GetKey ونزودها برمز المفتاح عن طريق القائمة KeyCode والتي تحوي رموز جميع المفاتيح. في كل مرة نستدعي ()Input.GetKey تقوم بفحص إذا ما كان اللاعب يضغط على المفتاح المطلوب، وتعطينا true في هذه الحالة، أو false في حال لم يكن اللاعب يضغط عليه. وباستخدام الجمل الشرطية if نقوم بربط حركة الكائن بالمفتاح المطلوب. لاحظ أننا قمنا باستخدام ()transform.Translate كما رأينا في الوحدة السابقة وذلك لتحريك الكائن على المحورين x و y مع ملاحظة استخدام قيمة سالبة للسرعة لتكون الحركة في الاتجاه السالب بالنسبة للحركة يسارا أو لأسفل.

قد نحتاج أحيانا لأن نقرأ المفتاح مرة واحدة فقط بدلا من قراءته بشكل مستمر. مثال على ذلك عملية القفز في ألعاب المنصات، حيث يحتاج اللاعب عادة لأن يقوم بالضغط مجددا على المفتاح ليقفز مرة أخرى، لا أن تتكرر عملية القفز بإبقاء مفتاح القفز مضغوطا. من أجل تحقيق ذلك يمكننا استخدام دّالّة أخرى وهي ()Input.GetKeyDown والتي تعطيك true مرة واحدة فقط عند الضغط على المفتاح لأول

مرة، ثم تعود لتعطي false مع استمرار الضغط، وبهذا تجبر اللاعب على رفع اصبعه عن المفتاح والضغط مجددا. جرب استبدال ()GetKey بـ ()GetKeyDown في السرد 4 ولاحظ الفرق.

إن كنت مبتدئا في البرمجة ولا تعرف الفرق بين استخدام if بشكل منفصل أو استخدام else if سأوضح فيما يلي الفرق بينهما مستعينا بالسرد 5: باستخدامنا للجملة الشرطية if بشكل منفصل لكل مفتاح، فإننا نسمح بقراءة أكثر من مفتاح في نفس الوقت ونسمح لكل منها أن يؤثر في حركة الكائن. أي أن اللاعب إذا ضغط السهمين الأعلى والأيمن فإن الكائن سيتحرك بشكل قطري لأعلى يمين الشاشة. أما إذا ضغط لأعلى وأسفل فإنهما سيلغيان تأثير بعضهما وسيبقى الكائن في مكانه. إذا أردنا أن نسمح فقط بمفتاح واحد دون الآخر على كل محور حركة (أفقيا أو عموديا)، يمكننا استعمال else if كي نمنع مثلا أن يأخذ قراءة جهتين متقابلتين في نفس الوقت، بحيث تعطى الأولوية للمفتاح صاحب الجملة الشرطية الأولى.

```
//حاول قراءة السهم الأعلى وفي حال لم يكن مضغوطا حاول قراءة الأسفل
if(Input.GetKey(KeyCode.UpArrow)){
     transform.Translate(0, speed * Time.deltaTime, 0);
} else if(Input.GetKey(KeyCode.DownArrow)){
     transform.Translate(0, -speed * Time.deltaTime, 0);
}

//حاول قراءة السهم الأيمن وفي حال لم يكن مضغوطا حاول قراءة الأيسر
if(Input.GetKey(KeyCode.RightArrow)){
     transform.Translate(speed * Time.deltaTime, 0, 0);
} else if(Input.GetKey (KeyCode.LeftArrow)){
     transform.Translate(-speed * Time.deltaTime, 0, 0);
}
```

السرد 5: استخدام else if لحصر القراءة بمفتاح واحد وتحديد تفضيل مفتاح على الآخر

ما ينطبق على الأسهم في المثال الذي ذكرته ينطبق على جميع المفاتيح الأخرى، كل ما عليك هو اختيار المفتاح الذي ترغب بقراءته من القائمة KeyCode. يمكنك الاطلاع على المشهد scene2 في المشروع المرفق لترى المثال الذي استعملناه في هذا الفصل كاملا.

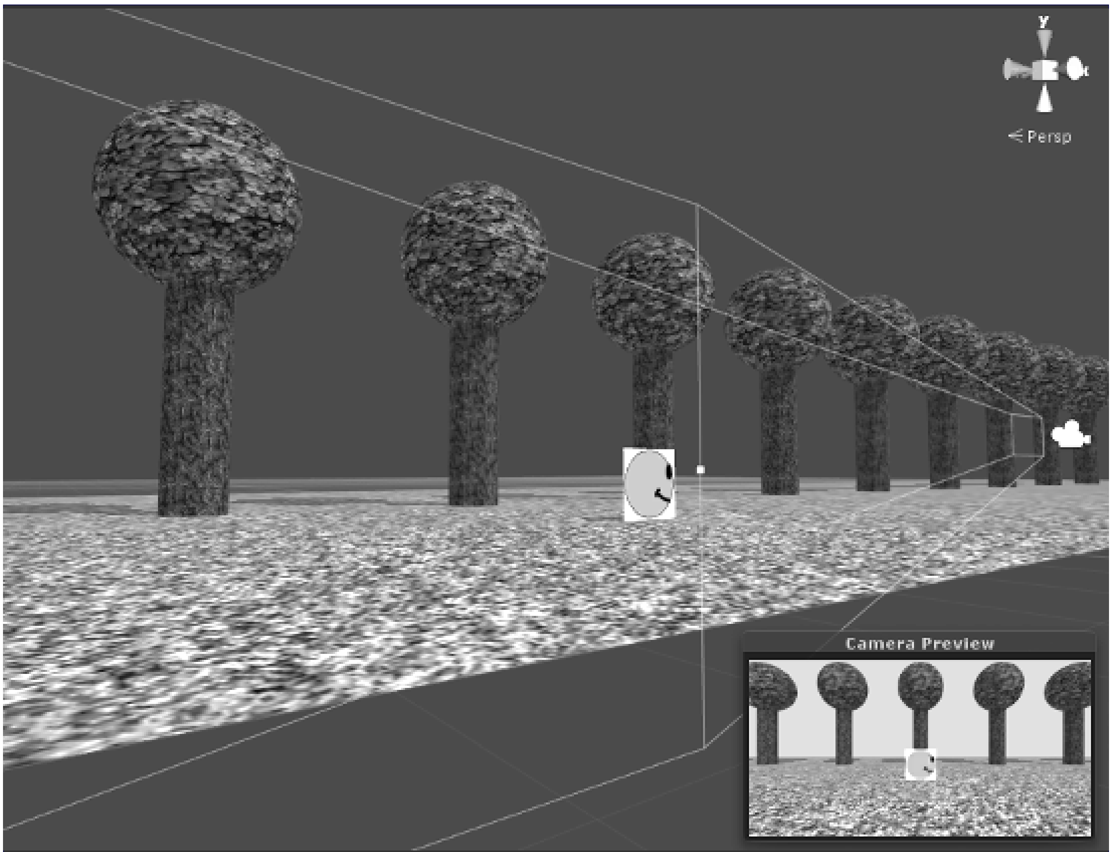
الفصل الثاني: تطبيق نظام إدخال ألعاب المنصات

تعتبر ألعاب المنصات من أكثر أنواع الألعاب ثنائية الأبعاد شهرة، بل وتعدت ذلك إلى الألعاب ثلاثية الأبعاد، فألعاب مثل Super Mario و Castlevania تعتبر من أشهر الألعاب القديمة التي تنتمي لهذا النوع، بالإضافة لعدد من الألعاب الحديثة الناجحة مثل Braid, FEZ, Super Meat Boy. تعتمد هذه الألعاب بشكل أساسي على ميكانيكية القفز للتحرك بين المنصات في اللعبة والتغلب على الخصوم وحل الألغاز، بالإضافة لبعض الميكانيكيات الأخرى مثل التصويب.

بما أننا الآن نستطيع قراءة مدخلات اللاعب عن طريق لوحة المفاتيح، يمكننا صنع نظام إدخال بسيط يعتمد على الأسهم للحركة ومفتاح المسافة للقفز. لكن وبما أننا لم نتعلم حتى الآن كيفية اكتشاف التصادمات لنعرف ما إذا كان اللاعب يقف على منصة أم أنه يجب أن يسقط للأسفل. سنكتفي باعتبار أنه يقف على الأرض طالما أن الإحداثي y لموقعه يساوي قيمة ثابتة.

شاهدنا في مثال سابق كيفية تطبيق الحركة إلى اليمين واليسار، لذا فالوظائف التي تنقصنا لتطبيق نظام الحركة المطلوب هي: الجاذبية الأرضية، والقفز، ومتابعة الكاميرا لشخصية اللاعب في حركتها يمينا أو يسارا.

قبل أن نبدأ في كتابة البريمجات الخاصة بنظام تحكم المنصات، علينا أولا أن نقوم ببناء مشهد بسيط لتطبيق التحكم داخله. قم ببناء مشهد مماثل للشكل17 مستخدما الأشكال الأساسية والعلاقات بين الكائنات والإكساءات المناسبة. لاحظ أننا قمنا ببناء شخصية اللاعب باستخدام الكائن Quad وذلك حتى تكون ثنائية الأبعاد.



الشكل 17: المشهد الخاص بتطبيق نظام إدخال ألعاب المنصات

الغرض من بناء هذه الخلفية البسيطة باستخدام الأشجار هو ملاحظة حركة الكاميرا حين نقوم بتطبيق

آلية تتبع اللاعب. لاحظ أنني قمت هنا باستخدام مشهد طولي ممتد على المحور x يمينا ويسارا بينما عمقه على المحور z أقل بكثير؛ وذلك نظرا لطبيعة الحركة الأفقية الغالبة على ألعاب المنصات. يمكن أن تقوم بعمل شيء أبسط إن استصعب عليك بناء مشهد كهذا رغم أنني أشجعك على المحاولة؛ وذلك لتتمرس أكثر على بناء المشاهد وخاصة اختيار قيم التكرار للإكساءات بما يتناسب مع قياس الكائنات.

لنقم الآن بكتابة البريمج الخاص بالتحكم بشخصية اللاعب، وهذا البريمج له عدة مهام: أولها أن يتأكد من تطبيق الجاذبية الأرضية بشكل صحيح على الكائن وذلك بسحبه باتجاه الأرض إن كان مرتفعا وعدم السماح له بالنزول تحت مستوى الأرض، وثاني هذه المهام هو التحكم بحركة الكائن من خلال قراءة مدخلات اللاعب. السرد 6 يوضح الكود الخاص بهذا البريمج، ولنسمّه PlatformerControl.

```
using UnityEngine;
using System.Collections;

public class PlatformerControl : MonoBehaviour {

	//السرعة العمودية عند بداية القفز
	public float jumpSpeed = 7;

	//سرعة السقوط
	public float gravity = 9.8f;

	//سرعة الحركة الأفقية
	public float movementSpeed = 5;

	//لتخزين السرعة الأفقية والعمودية قبل تحريك الكائن
	private Vector2 speed;

	// Use this for initialization
	void Start () {

	}

	// Update is called once per frame
	void Update () {
		//قراءة مدخلات اتجاه الحركة
		if(Input.GetKey(KeyCode.RightArrow)){
			speed.x = movementSpeed;
		} else if(Input.GetKey(KeyCode.LeftArrow)){
			speed.x = -movementSpeed;
		} else {
			speed.x = 0;
		}

		//قراءة مدخل القفز
		if(Input.GetKeyDown(KeyCode.Space)){
			//تطبيق القفز فقط في حالة كون الكائن يقف على الأرض
			if(transform.position.y == 0.5f){
				speed.y = jumpSpeed;
			}
		}
```

```
41.
42.             //تحريك الكائن
43.             transform.Translate(speed.x * Time.deltaTime,
44.                                 speed.y * Time.deltaTime,
45.                                 0);
46.
47.             //تطبيق الجاذبية الأرضية في حال القفز
48.             if(transform.position.y > 0.5f){
49.                   speed.y = speed.y - gravity * Time.deltaTime;
50.             } else {
51.                   speed.y = 0;
52.                   Vector3 newPosition = transform.position;
53.                   newPosition.y = 0.5f;
54.                   transform.position = newPosition;
55.             }
56.       }
57. }
```

السرد 6: بريمج تطبيق نظام إدخال المنصات

الملاحظة الأولى هي طول البريمج وتعقيده نسبيا مقارنة مع ما كنا نتعامل معه حتى الآن. هذا أمر طبيعي لكونه البريمج الأول الذي يقوم بوظيفة حقيقة متعلقة بتطوير الألعاب، والقادم أعظم! لعلك تدرك الآن ولو بشكل جزئي كمية العمل المطلوبة لصناعة لعبة حقيقية مهما كانت صغيرة وبسيطة.

لندخل الآن إلى البريمج ونشرحه بالتفصيل. مبدئيا لدينا ثلاث سرعات تتحكم بحركة الكائن (وهو في هذه الحالة شخصية اللاعب التي يتحكم بها). السرعة الأولى هي jumpSpeed وهي سرعة القفز. ما نعنيه بسرعة القفز هو سرعة الكائن العمودية نحو الأعلى في اللحظة التي يرتفع فيها عن الأرض. هذه السرعة تبدأ بالتناقص تدريجيا بمرور الوقت إلى أن تصل إلى الصفر عند الارتفاع الأقصى للقفزة، بعدها تبدأ بالتناقص إلى ما تحت الصفر مما يجعل الكائن يبدأ بالسقوط ثانية نحو الأسفل بفعل الجاذبية. السرعة الثانية هي gravity والتي تحدد سرعة السقوط نحو الأسفل. فكلما زدنا سرعة الجاذبية، زادت الوتيرة التي تتناقص بها سرعة القفز، مما يجعل السقوط أسرع وبالتالي ارتفاع القفزة أقل. سنناقش العلاقة بين السرعتين بتفصيل أكثر بعد قليل.

السرعة الثالثة وهي السرعة الأفقية movementSpeed وهي سرعة الحركة يمينا أو يسارا كما رأيناها في مثال في الفصل السابق. عندما نجمع هذه السرعات الثلاث تعطينا محصلة بسرعة متجهة لها قيمتان على المحورين x و y، لذا نقوم باستخدام المتغير speed من نوع Vector2 وهو متجه ثنائي الأبعاد لتخزين هاتين القيمتين واستخدامهما في كل دورة تصيير، خاصة أننا نحتاج لأن نحتفظ بقيمة السرعة العمودية التي تتغير بين تصيير كل إطارين وينبغي أن نحافظ على القيمة لاستخدامها في تصيير الإطار المقبل.

الخطوة الأولى في دورة التحديث (الأسطر 26 إلى 32) معروفة لدينا وهي قراءة مدخلات لوحة المفاتيح الخاصة بالحركة الأفقية، فالسهم الأيمن يعطينا قيمة للسرعة تساوي المتغير

movementSpeed، بينما السهم الأيسر يعطينا القيمة السالبة لنفس القيمة. أما في حالة عدم ضغط اللاعب لأي من مفتاحي السهمين فإن السرعة الأفقية تساوي صفرا. لاحظ أننا نخزن قيمة السرعة الأفقية في العضو x داخل المتغير speed وذلك لنستخدمه لاحقا في إزاحة الكائن في الاتجاه المطلوب.

الخطوة الثانية في دورة التحديث (الأسطر 35 إلى 40) هي قراءة مفتاح المسافة الخاص بالقفز. لاحظ أننا استخدمنا هنا ()GetKeyDown وذلك لنمنع القفز المتتالي باستمرار الضغط على مفتاح المسافة دون رفع اليد عنه. الملاحظة الثانية المهمة هي أنه لا يمكن لشخصية اللاعب أن تقفز ما لم تكن واقفة على الأرض. عند بناء المشهد الأصلي كان الإحداثي y لموقع كائن الشخصية هو 0.5 بالتالي فإن هذه القيمة تعتبر عندنا مرجعية لمعرفة ما إذا كانت الشخصية تقف على الأرض أم لا. في حالة تحققنا من كون الشخصية تقف على الأرض، نقوم بتغيير السرعة العمودية لتصبح مساوية لقيمة jumpSpeed ونقوم بتخزين هذه القيمة في العضو y داخل المتغير speed.

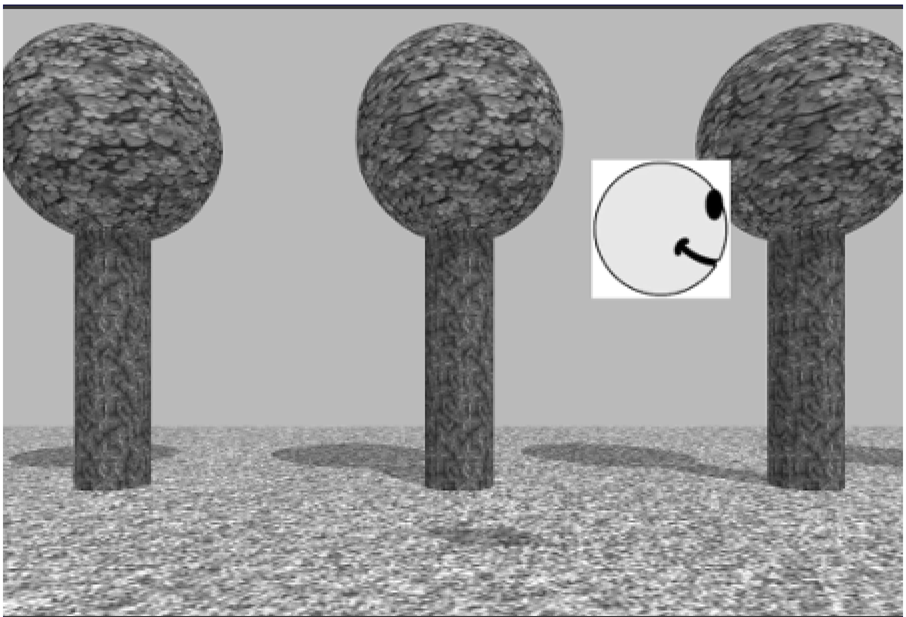
الخطوة الثالثة (الأسطر 43 إلى 45) هي أن نقوم بتنفيذ الإزاحة حسب قيم السرعة التي سبق وقمنا بحسابها اعتمادا على مدخلات اللاعب. وذلك عن طريق الدّالّة ()transform.Translate كما تعودنا. الحركة ستكون على المحورين x و y حسب القيم المخزنة داخل المتغير speed ونضربها بالزمن المنقضي كما سبق وشرحنا ذلك.

بعد أن انتهينا من تحريك الكائن بقي علينا خطوة أخيرة، وهي حساب السرعة العمودية الجديدة للاعب. كما سبق وذكرنا، فإن القيمة 0.5 على المحور y بالنسبة لموقع الكائن تعني أن الشخصية تقف على الأرض. فإذا كانت القيمة أكبر من ذلك (الأسطر 48 إلى 50)، فإن ذلك يدل على أن الكائن في الهواء حاليا، مما يستدعي أن نقوم بتقليل سرعته العمودية اعتمادا على الجاذبية. لاحظ أن قيمة السرعة العمودية تتناقص في كل دورة تحديث بمقدار يساوي سرعة الجاذبية مضروبة في الزمن المنقضي. وبما أن السرعة تتناقص، فإن مقدار الإزاحة العمودية في دورة التحديث المقبلة ستكون أقل، وهكذا حتى تصل إلى الصفر. يمكن ملاحظة ذلك بتناقص سرعة ارتفاع الكائن لأعلى بمرور الزمن حتى يتوقف في الهواء ثم يعود للسقوط بسرعة تبدأ قليلة، ثم تزيد بمرور الزمن.

القسم الآخر من الخطوة الأخيرة (الأسطر 50 إلى 55) يتعلق بكون شخصية اللاعب في وضع غير وضع القفز، وهنا نحن أمام احتمالين: إما أن تكون قيمة الموقع على المحور y تساوي 0.5 وهي القيمة الأولية، وإما أن تكون قد نقصت لما دون تلك القيمة نتيجة للإزاحة في الخطوة الثالثة. وبما أن الإزاحة تعتمد على الزمن المنقضي والذي لا يمكننا التحكم فيه، فلا يمكننا أن نضمن عدم الحصول على قيم أقل من 0.5؛ لأننا – كما ذكرت في مقدمة هذا الفصل – لا نتعامل حتى اللحظة مع اكتشاف التصادمات بين الكائنات، بالتالي فإن الأرضية لن تمنع إزاحة الكائن لأسفل. وحتى نكون في الجانب الآمن، فإننا نقوم بإرجاع السرعة العمودية إلى الصفر حتى نضمن بقاء الجسم على الأرض حتى القفزة القادمة، إضافة إلى التأكد من وضع الكائن في موقعه الأصلي على المحور y وهو 0.5.

لا يسمح لك Unity بتغيير عضو واحد من أعضاء موقع الكائن transform.position بشكل مباشر، فلن يقبل مثلا الأمر ; transform.position.y = 0.5f. لذا عليك أولا أن تقوم بتخزين قيمة الموقع كاملة في متغير من نوع Vector3 ومن ثم تعدل على الأعضاء داخل هذا المتغير بما يناسبك، ثم بعد ذلك تعيد قيمة المتغير إلى transform.position كما في الأسطر 52 إلى 54 في السرد 6

بعد أن تقوم ببناء المشهد كما في الشكل 17 (أو أي مشهد مماثل، بشرط وجود كائنات في الخلفية لتمييز حركة اللاعب والكاميرا) وتضيف البريمج PlatformerControl على كائن شخصية اللاعب، قم بتشغيل اللعبة وتجربة التحرك والقفز. الشكل 18 يوضح لقطة من اللعبة أثناء القفز.


الشكل 18: تطبيق نظام تحكم المنصات وتنفيذ القفز

بقي الآن أن نقوم بتحريك الكاميرا لكي تتبع اللاعب، ذلك أنه حتى اللحظة يمكن أن تغادر شخصية اللاعب مجال رؤية الكاميرا فتصبح غير مرئية مما يجعل اللعب مستحيلا. يمكننا القيام بذلك بطريقتين: الطريقة الأولى والأسهل هي أن تضيف كائن الكاميرا كابن لكائن شخصية اللاعب. بالتالي ستتحرك الكاميرا معه أينما تحرك أفقيا وعموديا. في هذه الحالة ستلاحظ أن كائن الشخصية سيبقى ثابتا بالنسبة لنافذة اللعبة طول الوقت. الطريقة الثانية والأكثر تقدما هي كتابة بريمج يسمح للكاميرا بتتبع شخصية اللاعب، ويعطي هامشا يمكن من خلاله للشخصية أن تتحرك لمسافة داخل الشاشة قبل أن تبدأ الكاميرا بتتبّعها، ويمكن تغيير هذا الهامش بشكل ديناميكي.

لنناقش ببعض التفصيل الطريقة الثانية، والتي ستجعل نظام التحكم النهائي يبدو أكثر احترافية وأكثر متعة للاعب. في بداية اللعبة تكون الشخصية في منتصف مجال الرؤية بالنسبة للكاميرا، ثم تبدأ بالتحرك نحو اليمين مثلا. عندما تصبح الشخصية على مسافة معينة على المحور x عن الكاميرا ستبدأ

حينها الكاميرا بالتحرك نحو اليمين لتتبعها. السرد 7 يوضح كيفية تطبيق آلية التتبع المذكورة. قم بإنشاء البريمج PlayerTracking ثم أضفه إلى كائن الكاميرا.

```
using UnityEngine;
using System.Collections;

public class PlayerTracking : MonoBehaviour {
	//نحتاج مرجعا لأحد مكوّنات كائن شخصية اللاعب حتى نتمكن من معرفة موقعه
	public Transform playerCharacter;
	//أقصى مسافة للحركة نحو اليمين قبل أن تبدأ الكاميرا بالتتبع
	public float maxDistanceRight = 1.5f;
	//أقصى مسافة للحركة نحو اليسار قبل أن تبدأ الكاميرا بالتتبع
	public float maxDistanceLeft = 1.5f;
	//أقصى مسافة للحركة نحو الأعلى قبل أن تبدأ الكاميرا بالتتبع
	public float maxDistanceUp = 1.0f;
	//أقصى مسافة للحركة نحو الأسفل قبل أن تبدأ الكاميرا بالتتبع
	public float maxDistanceDown = 1.0f;

	// Use this for initialization
	void Start () {

	}

	// هذه المرة قمنا باستخدام الدّالّة LateUpdate
	void LateUpdate () {
		//موقع الكاميرا الحالي
		Vector3 camPos = transform.position;
		//موقع اللاعب الحالي
		Vector3 playerPos = playerCharacter.position;

		//هل اللاعب إلى يمين الكاميرا بمسافة تتجاوز الحد الأقصى؟
		if(playerPos.x - camPos.x > maxDistanceRight){
			camPos.x = playerPos.x - maxDistanceRight;
		}
		//هل اللاعب إلى يسار الكاميرا بمسافة تتجاوز الحد الأقصى؟
		else if(camPos.x - playerPos.x > maxDistanceLeft){
			camPos.x = playerPos.x + maxDistanceLeft;
		}

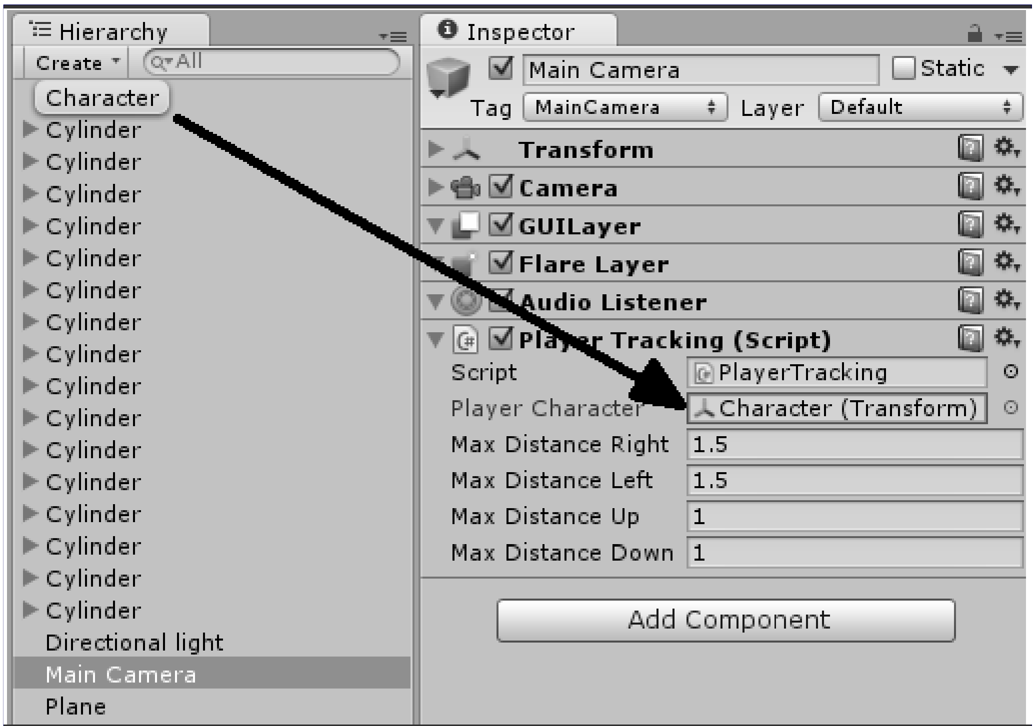
		//هل اللاعب أعلى الكاميرا بمسافة تتجاوز الحد الأقصى؟
		if(playerPos.y - camPos.y > maxDistanceUp){
			camPos.y = playerPos.y - maxDistanceUp;
		}
		//هل اللاعب أسفل الكاميرا بمسافة تتجاوز الحد الأقصى؟
		else if(camPos.y - playerPos.y > maxDistanceDown){
			camPos.y = playerPos.y + maxDistanceDown;
		}
		//تحديث موقع الكاميرا بعد إجراء التعديلات
		transform.position = camPos;
	}
}
```

السرد 7: آلية تتبع الكاميرا لشخصية اللاعب

هناك الكثير من الأمور الجديدة في هذا البريمج والتي ينبغي أن نتحدث عنها ببعض التفصيل. أول هذه الأمور هو المتغير playerCharacter من نوع Transform والذي يختلف عن المتغيرات التي تعاملنا معها حتى الآن وكان معظمها رقميا، بحيث يمكننا أن ندخل قيمتها باستخدام لوحة المفاتيح.

نظرا لطبيعة عمل هذا البريمج، فإنه يتعامل مع أكثر من كائن، فمن ناحية نحن نضيفه إلى كائن الكاميرا لكي يقوم بتحريكه، ومن ناحية أخرى يجب أن يعرف موقع اللاعب حتى يقوم بمهمة تتبعه. لذا قمنا بتعريف المتغير playerCharacter حتى يكون مرجعا لكائن اللاعب، وتحديدا للمكون Transform في هذا الكائن. بقي فقط أن نربط كائن اللاعب مع هذا المتغير حتى نتمكن من معرفة مكان اللاعب عند تشغيل اللعبة. الشكل 19 يوضح شاشة الخصائص بعد ربط الكائن مع المتغير داخل البريمج.

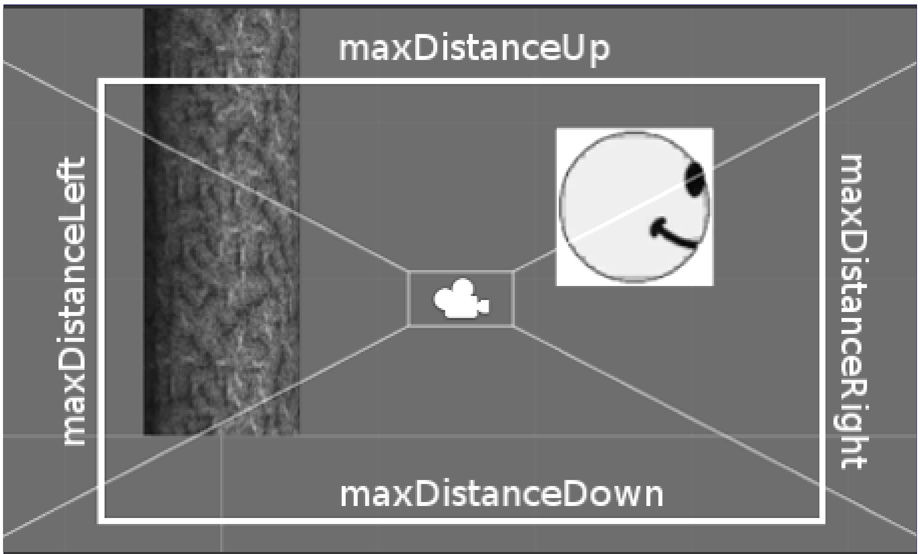
لتربط كائنا من المشهد مع متغير داخل بريمج، قم بسحب الكائن من هرمية المشهد إلى داخل الخانة الخاصة بالمتغير في شاشة الخصائص التي يظهر فيها البريمج. في حالة ربط كائن الشخصية مع بريمج PlayerTracking، قم أولا باختيار الكاميرا من المشهد، ثم قم بإضافة البريمج لها إن لم تكن أضفته بعد. عند إضافة البريمج ستظهر الخانة Player Character كما في الشكل 19، كل ما عليك هو سحب كائن شخصية اللاعب إلى تلك الخانة



الشكل 19: كيفية ربط الكائن مع المتغير داخل البريمج

إضافة إلى المتغير playerCharacter، قمنا بتعريف أربع متغيرات لتحديد الإطار الذي يمكن للاعب التحرك في داخله دون أن تتبعه الكاميرا. هذه المتغيرات تحمل أسماء تدل على موقع اللاعب بالنسبة للكاميرا على المحورين x و y. فلدينا maxDistanceRight و maxDistanceLeft اللذين يحددان أقصى

مسافة إلى يمين ويسار الكاميرا، فإذا كانت شخصية اللاعب على يمين الكاميرا بمسافة تساوي maxDistanceRight، فإن الكاميرا تتحرك باتجاه اليمين حتى لا تسمح للمسافة بالزيادة عن هذا الحد. كذلك الأمر بالنسبة للمتغيرين maxDistanceUp و maxDistanceDown مع حركة اللاعب على المحور y. الشكل 20 يوضح كيف سيبدو شكل هذه المحددات لو رسمناها على شكل مستطيل حول موقع الكاميرا.



الشكل 20: المساحة التي يمكن أن يتحرك كائن شخصية اللاعب داخلها دون أن تتحرك الكاميرا

بعد أن قمنا بتحديد هذه المسافات علينا أن نتتبع موقع اللاعب في كل دورة تصيير حتى نعرف إن كان ينبغي أن نحرك الكاميرا في اتجاه ما. بداية لاحظ أننا في السطر 22 استخدمنا الدّالّة ()LateUpdate بدلا من ()Update التي اعتدنا استخدامها. وجه الشبه بينهما أنهما تستدعيان في كل دورة تحديث، أما الفرق بينهما أن Unity يقوم باستدعاء ()Update أولا من جميع البريمجات الموجودة في المشهد، ومن ثم يعود ليستدعي ()LateUpdate.

تذكر أن المشهد الحالي يحتوي على بريمجين هما PlatformerControl الذي يقوم بقراءة مدخلات اللاعب وتحريك الشخصية، و PlayerTracking الذي يساعد الكاميرا على تتبع موقع اللاعب. ما نريد أن نضمنه هو أن تحريك شخصية اللاعب يتم أولا ثم يتم بعد ذلك تحريك الكاميرا. الطريقة الأسهل لذلك هي أن نقوم بتحديث بريمج الكاميرا عن طريق ()LateUpdate مما يضمن لنا أنه سيتم تحديثه بعد أن تكون شخصية اللاعب قد تحركت وأصبحت في موقعها الجديد الذي نريد أن نقارنه بموقع الكاميرا. ربما لن تلاحظ فرقا إن قمت باستخدام ()Update بدلا من ()LateUpdate لأنك قد تكون محظوظا ويقوم Unity بتنفيذ PlatformerControl قبل PlayerTracking، لكن في عالم البرمجة يجب ألا تترك شيئا

للصدفة وعليك دائما أن تحسب الاحتمال الأسوأ وتحتاط منه.

كما ترى في السطرين 24 و 26، فإننا نبدأ بتخزين موقعي كل من اللاعب والكاميرا حتى نقوم بعمل الحسابات المطلوبة بينهما. نبدأ بعد ذلك في فحص الاحتمالات الممكنة على المحور x وذلك في الأسطر 30 إلى 36. آخذين في الحسبان أن الاتجاه الموجب للمحور x هو إلى جهة اليمين والاتجاه السالب إلى جهة اليسار، نقوم بطرح الإحداثي x لموقع الكاميرا من الإحداثي x لموقع اللاعب، فإذا كان الناتج أكبر من المسافة المحددة من جهة اليمين، فإننا نقوم بتغيير موقع الكاميرا لتتبع اللاعب.

المهم هو كيف نحسب الموقع الجديد للكاميرا على المحور x. المثال التالي يوضح الطريقة: لنفرض أن أقصى مسافة مسموح بها من جهة اليمين هي 1.5 كما في السطر 8، ولنفرض أن شخصية اللاعب تحركت لليمين في دورة التحديث الحالية ووصل للموقع 1.6 بينما لا تزال الكاميرا في الموقع 0. لو حسبنا الفرق بين الموقعين سيكون 1.6 = 0 – 1.6، وهي قيمة أكبر من القيمة المسموح بها وهي 1.5 لذا يجب أن نحرك الكاميرا مع مراعاة أن تبقى شخصية اللاعب على مسافة 1.5 إلى يمين الكاميرا، وذلك لضمان استمرار حركة الكاميرا مع حركة الشخصية نحو اليمين. لذا نقوم بطرح المسافة القصوى المسموحة من موقع الشخصية الحالي 0.1 = 1.5 – 1.6 وهو الموقع الجديد الذي يجب أن ننقل الكاميرا إليه، وهذا بالضبط ما نفعله في السطر 30.

هذه الطريقة تستخدم أيضا في حالة كانت شخصية اللاعب إلى يسار الكاميرا مع تغيير بسيط، وهو أننا نضيف أقصى مسافة مسموح بها من جهة اليسار إلى موقع الشخصية الحالي لحساب موقع الكاميرا الجديد كما في السطر 34. وكذلك الأمر بالنسبة لحركة اللاعب على المحور y والمسافة المسموح بها من أعلى وأسفل.

بعد الانتهاء من حساب الموقع الجديد للكاميرا بناء على موقع اللاعب وحدود الحركة التي قمنا بضبطها، تبقى الخطوة الأخيرة وهي اعتماد الموقع الجديد للكاميرا بوضع قيمته في transform.position كما في السطر 46. يمكنك مشاهدة النتيجة النهائية لهذا العمل في المشهد scene3 في المشروع المرفق.

الفصل الثالث: قراءة مدخلات الفأرة

بعد أن تعرفنا على كيفية قراءة واستخدام مدخلات لوحة المفاتيح، لننتقل إلى أداة الإدخال الرئيسية الثانية في الحواسيب الشخصية وهي الفأرة. لا يكاد يغفل أي متابع لألعاب الكمبيوتر عن أهمية هذه الأداة في الألعاب، فلها الدور الأكبر في ألعاب التصويب، كما أنها تستخدم لإعطاء الكثير من الأوامر في الألعاب الاستراتيجية وكذلك في واجهة المستخدم كانتقاء الأدوات والتفاعل مع الحوارات.

ما يهمنا في هذا الفصل هو التعرف على كيفية قراءة حركة الفأرة على المحورين الأفقي والعمودي،

إضافة إلى قراءة حركات النقر على الأزرار الثلاث و تدوير عجلة الفأرة. لنبدأ بمشهد جديد يحتوي على كائن واحد عبارة عن كرة تتوسط المشهد، تماما كما فعلنا في الفصل الأول من هذه الوحدة. سنقوم بإضافة بريمج لهذه الكرة بحيث يعمل على تحريكها أفقيا وعموديا حسب حركة الفأرة، كما تقوم بتغيير لونها عند النقر على أزرار الفأرة الثلاث، وأخيرا سنقوم بتغيير حجم الكرة حسب استدارة عجلة الفأرة (ملاحظة: من الأفضل أن تقوم بإبعاد الكاميرا نحو الخلف حتى لا تخرج الكرة خارج مدى الرؤية بسهولة، ولتكن في الموقع (0, 0, -70) مثلا). السرد 8 يوضح الكود الخاص بقراءة مدخلات الفأرة وترجمتها للسلوك المطلوب.

```
using UnityEngine;
using System.Collections;

public class MouseMovement : MonoBehaviour {

	//سرعة حركة الكائن
	public float movementSpeed = 5;

	// الألوان الخاصة بالأزرار
	public Color left = Color.red;
	public Color right = Color.green;
	public Color middle = Color.blue;

	//معامل الزيادة أو النقصان في الحجم عند تدوير العجلة
	public float scaleFactor = 1;

	//موقع مؤشر الفأرة في الإطار السابق
	//نحتاجه لقياس قيمة الإزاحة
	Vector3 lastMousePosition;

	void Start () {
		//لجعل الإزاحة تساوي صفراً عند البداية
		lastMousePosition = Input.mousePosition;
	}

	void Update () {

		if(Input.GetMouseButton(0)){
			//الزر الأيسر
			renderer.material.color = left;
		} else if(Input.GetMouseButton(1)){
			//الزر الأيمن
			renderer.material.color = right;
		} else if(Input.GetMouseButton(2)){
			//الزر الأوسط
			renderer.material.color = middle;
		}

		//حساب قيمة إزاحة المؤشر
		Vector3 mouseDelta = Input.mousePosition - lastMousePosition;
		transform.Translate(
			movementSpeed * Time.deltaTime * mouseDelta.x,
```

```
43.                     movementSpeed * Time.deltaTime * mouseDelta.y, 0);
44.
45.             //تحديث الموقع الأخير استعدادا للإطار القادم
46.             lastMousePosition = Input.mousePosition;
47.             //قراءة استدارة العجلة
48.             float wheel = Input.GetAxis("Mouse ScrollWheel");
49.             if(wheel > 0){
50.                     //استدارة العجلة نحو الأعلى
51.                     transform.localScale += Vector3.one * scaleFactor;
52.             } else if(wheel < 0){
53.                     //استدارة العجلة نحو الأسفل
54.                     transform.localScale -= Vector3.one * scaleFactor;
55.             }
56.     }
57. }
```

السرد 8: قراءة مدخلات الفأرة

لنناقش أهم ما يحتويه هذا البريمج، لدينا - كما رأينا سابقا - سرعة الحركة في السطر 7 بالإضافة إلى ثلاث متغيرات من نوع Color في الأسطر 10 إلى 12 والتي تقوم بتخزين قيمة لون معين يمكن اختيارها من لوح الألوان الذي يظهر في شاشة الخصائص أو إعطائها قيمة ثابتة كما فعلنا. في السطر 15 قمنا بتعريف المتغير lastMousePosition بهدف تخزين آخر موقع للمؤشر حتى نستخدمه في حساب الإزاحة عند تصيير كل إطار. عند بداية التشغيل (السطر 23) نستعمل lastMousePosition لتخزين قيمة الموقع الحالي للمؤشر والتي نحصل عليها عن طريق Input.mousePosition وذلك حتى تكون الإزاحة صفراً عند تصيير الإطار الأول.

بعد ذلك ندخل في دورة التحديث ونقوم بقراءة المدخلات بشكل متتابع. البداية مع الأسطر من 28 إلى 37 والتي نفحص من خلالها ما إذا كان اللاعب يضغط حاليا على أحد الأزرار الثلاثة. الدّالّة ()Input.GetMouseButton تقوم بهذه المهمة وما علينا سوى تزويدها برقم الزر الذي نريد فحص حالته: 0 للزر الأيسر و 1 للأيمن و 2 للأوسط. ما نفعله في هذه الأسطر ببساطة هو تغيير لون الخامّة الحالية للكائن بناء على الزر الذي ضغطه اللاعب.

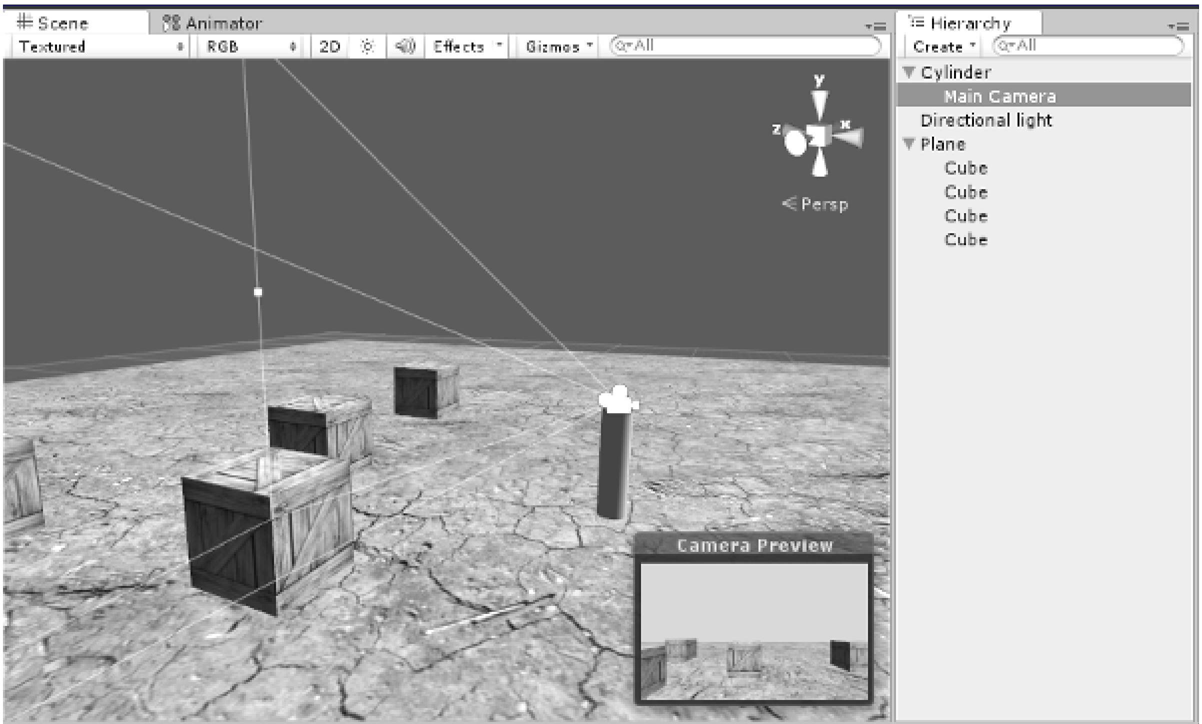
في الأسطر 40 إلى 43 نقوم بحساب المسافة التي تحركها المؤشر، وذلك بطرح موقعه خلال الدورة السابقة من موقعه الحالي، ثم نقوم بتحريك الكائن على المحورين x و y بمقدار المسافة المحسوبة مضروبة بسرعة الحركة. بما أننا نتحدث عن مؤشر الفأرة فإن موقعه بطبيعة الحال ثنائي الأبعاد، لذا فالقيمة z لموقعه تكون دائما صفراً ولا داعي لأن ندخلها في حساباتنا. بعد ذلك وفي السطر 46 نقوم بتحديث آخر موقع للمؤشر ليصبح مساويا لموقعه الحالي، بالتالي نصبح جاهزين لحساب الإزاحة الجديدة في الدورة القادمة.

الخطوة الأخيرة هي قراءة حركة عجلة الفأرة نحو الأعلى أو الأسفل، وذلك عن طريق الدّالّة ()GetAxisInput. والتي نعطيها اسم المحور المطلوب قراءته. يتم إعداد المحاور في نافذة خاصة

بالمدخلات قد نتطرق لها لاحقا. لكننا على الأقل نمتلك المحور الافتراضي المسمى Mouse ScrollWheel والمزود تلقائيا من قبل Unity وما علينا سوى استدعاء اسمه. إن لم تتحرك العجلة فإن القيمة المرجعة تساوي صفراً. أما إذا أدار اللاعب العجلة نحو الأسفل فإن القيمة تساوي 1-، وإذا أدارها نحو الأعلى فإن القيمة تساوي 1. بناء على هذه القيمة نقوم بإضافة أو طرح متجه بقيمة 1 مضروبا في قيمة معامل تغيير الحجم ونخزن القيمة في transform.localScale لتطبيقها على حجم الكائن. يمكنك الاطّلاع على هذا المثال في المشهد scene4 في المشروع المرفق.

الفصل الرابع: تطبيق نظام إدخال ألعاب منظور الشخص الأول

ألعاب منظور الشخص الأول first person games من أكثر الألعاب ثلاثية الأبعاد شعبية، وتنتمي لهذه الفئة الكثير من الألعاب الشهيرة في عالم ألعاب الفيديو مثل Doom و Half-Life و Call of Duty.
تعتمد هذه الألعاب على وضع الكاميرا مكان وجه اللاعب ورؤية عالم اللعبة من خلال عينيه مستخدمة الفأرة لتوجيه نظر اللاعب. إضافة للفأرة تستخدم عادة مفاتيح WSAD للتحرك في الاتجاهات الأربع.



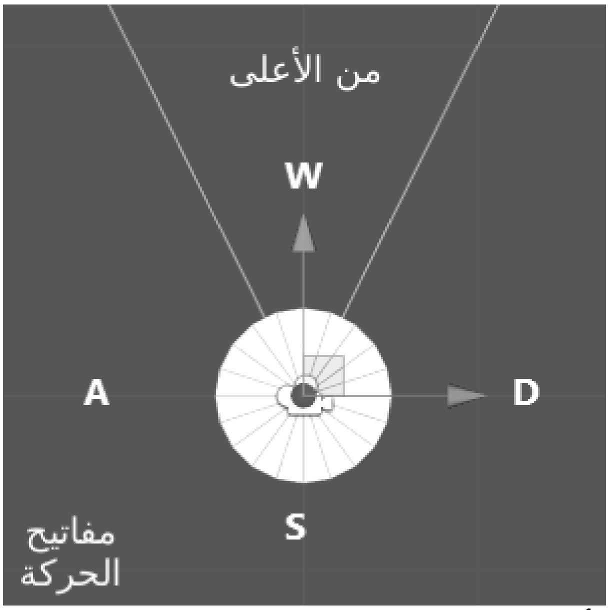
الشكل 21: كائن الأسطوانة الخاص بنظام إدخال منظور الشخص الأول

سنقوم في هذا الفصل بتطبيق هذا النوع من أنظمة الحركة باستخدام ما تعلمناه حتى الآن. بداية علينا أن نعرف كيف يتم بناء الكائن الخاص باللاعب في هذا النوع من الألعاب. بما أن شخصية اللاعب ليست مرئية في هذا النوع من الألعاب، سنكتفي بأسطوانة بطول مترين أي وحدتين في Unity، وجدير بالذكر أن هذا هو الحجم الافتراضي لكائن الأسطوانة في Unity. بما أن الكاميرا في مكان عيون اللاعب، يجب

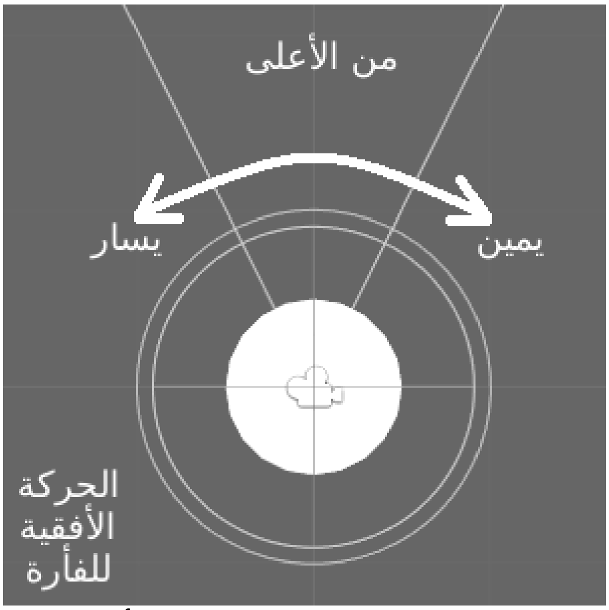
أن تتم إضافتها كابن لهذه الأسطوانة حتى تتحرك وتستدير معها. إضافة لذلك علينا أن نضيف أرضية وربما بعض الكائنات المرئية الأخرى لنصنع مشهدا يمكننا التحرك داخله كما في الشكل 21.

لاحظ أن الكاميرا مضافة كابن للأسطوانة وموقعها فوق سطحها العلوي، بينما الأسطوانة نفسها تقف فوق سطح الأرض. آلية توجيه النظر بالفأرة ستكون كما يلي: عندما يحرك اللاعب الفأرة بشكل أفقي، سنقوم بإدارة الأسطوانة حول محور المحور y مما سيجعل اللاعب يستدير إلى اليمين أو اليسار حسب اتجاه حركة الفأرة. أما عندما يحرك اللاعب الفأرة بشكل عمودي، سنقوم بإدارة الكاميرا فقط نحو الأعلى أو الأسفل حسب اتجاه الحركة. بهذا الشكل يصبح لدينا محوران مستقلان للاستدارة الأفقية أو العمودية، بنظام أشبه بحامل كاميرا التصوير ثلاثي الأرجل Tripod.

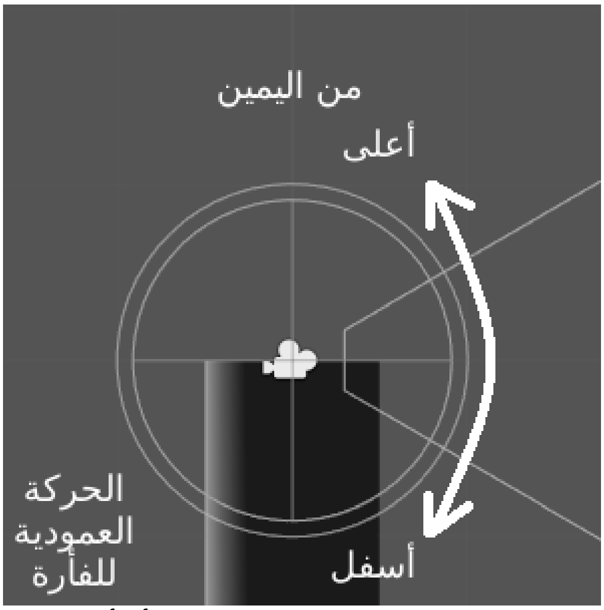
أما بالنسبة لحركة اللاعب، فضغط المفتاح W سيقوم بتحريك اللاعب نحو الأمام في الاتجاه الذي ينظر إليه حاليا (أي الاتجاه الموجب للمحور z الخاص بكائن الأسطوانة)، بينما يحركه المفتاح S عكس هذا الاتجاه. كذلك الأمر بالنسبة للجانبين حيث يحركه المفتاح D في الاتجاه الموجب للمحور x الخاص بالكائن أي نحو اليمين، بينما يحركه المفتاح A نحو اليسار. الشكل 22 يوضح كيفية استجابة كائن الأسطوانة والكاميرا لمدخلات اللاعب.



(أ) تتحرك الأسطوانة كاملة في الاتجاهات الأربع



(ب) تستدير الأسطوانة كاملة يمينا أو يسارا



(ت) تستدير الكاميرا فقط لأعلى أو أسفل

الشكل 22: تأثير مدخلات اللاعب على حركة ودوران كائن الأسطوانة والكاميرا

بقياس هذه الحركة على جسم إنسان، يمكننا الافتراض بأن مفاتيح الحركة تحرك الجسم كاملا، وبأن الحركة الأفقية للفأرة تقوم بإدارة الجسم كاملا نحو اليمين أو اليسار. أما في حالة الحركة العمودية للفأرة، فإن الرأس فقط هو الذي يتحرك ناظرا لأعلى أو لأسفل، مما ينتج الحركة النهائية بالشكل المطلوب.

بقي أن نذكر أن دوران الكاميرا على محورها المحلي x (أي الاستدارة العمودية) يجب أن يكون محددا

بين قيمتين عظميين للأعلى والأسفل، فلا يجب أن نسمح للكاميرا بالاستدارة إلى أن تصبح مقلوبة رأسا على عقب. لذلك علينا أن نحدد زاوية بين الكاميرا والأفق، ولتكن مثلا 60، بحيث لا نسمح لها بالنظر لأعلى أو لأسفل بمقدار يزيد عن هذه الزاوية.

البريمج FirstPersonControl الموضّح في السرد 9 يقوم بكل هذه العمليات التي ذكرتها، حيث يجب أن تتم إضافته إلى كائن الأسطوانة، والتي تحتوي بدورها على كائن الكاميرا كابن لها كما هو موضّح في الشكل 21 .

```
using UnityEngine;
using System.Collections;

public class FirstPersonControl : MonoBehaviour {

	//السرعة العمودية عند بداية القفز
	public float jumpSpeed = 0.25f;

	//سرعة السقوط
	public float gravity = 0.5f;

	//سرعة الحركة الأفقية على الأرض
	public float movementSpeed = 15;

	//سرعة النظر بالفأرة على المحورين الأفقي والعمودي
	public float horizontalMouseSpeed = 0.9f;
	public float verticalMouseSpeed = 0.5f;

	//أكبر قياس مسموح به لزاوية الدوران العمودي للكاميرا
	public float maxVerticalAngle = 60;

	//تخزين سرعة اللاعب على المحاور الثلاث لتنفيذ الحركة
	private Vector3 speed;

	//موقع الفأرة خلال الإطار السابق
	//ضروري لحساب الإزاحة
	private Vector3 lastMousePosition;

	//متغير لتخزين كائن الكاميرا
	private Transform camera;

	void Start () {
		lastMousePosition = Input.mousePosition;
		//البحث عن كائن الكاميرا في قائمة الأبناء
		camera = transform.FindChild("Main Camera");
	}

	void Update () {
		//الخطوة الأولى هي إدارة الأسطوانة عموديا على المحور y
		//بناء على الإزاحة الأفقية للفأرة يمينا أو يسارا
		Vector3 mouseDelta = Input.mousePosition - lastMousePosition;

```

```
		transform.RotateAround(
				Vector3.up, //محور الدوران
				mouseDelta.x *
				horizontalMouseSpeed *
				Time.deltaTime);//زاوية الدوران

		//الدوران العمودي الحالي لكائن الكاميرا
		float currentRotation = camera.localRotation.eulerAngles.x;

		//تحويل دوران الكاميرا العمودي من المدى بين 0 و 360
		//إلى المدى بين -180 و 180
		if(currentRotation > 180){
			currentRotation = currentRotation - 360;
		}

		//حساب الدوران خلال الإطار الحالي
		float ang =
			-mouseDelta.y * verticalMouseSpeed * Time.deltaTime;

		//الخطوة الثانية هي الدوران العمودي للكاميرا على محورها المحلي x
		//بناء على الإزاحة العمودية للفأرة لأعلى أو لأسفل
		//قبل ذلك يجب أن نتأكد من حدود الدوران العمودية المسموحة
		if((ang < 0 && ang + currentRotation > -maxVerticalAngle) ||
			(ang > 0 && ang + currentRotation < maxVerticalAngle)){
			camera.RotateAround(camera.right, ang);
		}

		//تحديث موقع الفأرة الأخير استعدادا للإطار المقبل
		lastMousePosition = Input.mousePosition;

		//الخطوة الثالثة هي تحديث الحركة
		if(Input.GetKey(KeyCode.A)){
			//تحرك لليسار
			speed.x = -movementSpeed * Time.deltaTime;
		} else if(Input.GetKey(KeyCode.D)){
			//تحرك لليمين
			speed.x = movementSpeed * Time.deltaTime;
		} else {
			speed.x = 0;
		}

		if(Input.GetKey(KeyCode.W)){
			//تحرك للأمام
			speed.z = movementSpeed * Time.deltaTime;
		} else if(Input.GetKey(KeyCode.S)){
			//تحرك للخلف
			speed.z = -movementSpeed * Time.deltaTime;
		} else {
			speed.z = 0;
		}

		//قراءة مدخل القفز
		if(Input.GetKeyDown(KeyCode.Space)){
			//تنفيذ القفز فقط في حال كون اللاعب يقف على الأرض
```

```
97.                 if(transform.position.y == 2.0f){
98.                       speed.y = jumpSpeed;
99.                 }
100.          }
101.
102.          //تحريك الأسطوانة حسب الاتجاهات المحسوبة
103.          transform.Translate(speed);
104.
105.          //تنفيذ الجاذبية في حال القفز
106.          if(transform.position.y > 2.0f){
107.                speed.y = speed.y - gravity * Time.deltaTime;
108.          } else {
109.                speed.y = 0;
110.                Vector3 newPosition = transform.position;
111.                newPosition.y = 2.0f;
112.                transform.position = newPosition;
113.          }
114.    }
115. }
```

السرد 9: تطبيق نظام تحكم منظور الشخص الأول

قد تبدو لك بعض الأجزاء من هذا البريمج مألوفة كونك سبق ورأيتها في السرد 6 عندما قمنا ببناء نظام تحكم ألعاب المنصات (إن لم تكن قرأت ذلك الفصل يمكنك العودة للشرح في الصفحة 30). لنناقش الآن الأجزاء الجديدة التي تخص نظام تحكم منظور الشخص الأول.

بعد أن قمنا بعريف بعض المتغيرات للتحكم بالحركة، استدعينا الدّالّة ()transform.FindChild في السطر 35 وذلك لنبحث في أبناء الكائن الحالي عن كائن محدد بالاسم. الاسم الذي قمنا بالبحث عنه في هذه الحالة هو Main Camera وهو الاسم الافتراضي المعطى من قبل Unity لكائن الكاميرا. وبما أننا سبق وأضفنا الكاميرا كابن للأسطوانة التي تحمل هذا البريمج، فإن الدّالّة المذكورة ستقوم بإيجاد الكاميرا وإرجاعها لنا ليتم تخزينها في المتغير camera مما يمكّننا من التعامل معها لاحقا.

في الخطوة الأولى في الأسطر من 43 إلى 47 قمنا باستدعاء الدّالّة ()transform.RotateAround ومهمتها تدوير الكائن حول محور محدد. فقمنا بتزويدها بكل من المحور الذي نريد أن ندير الكائن حوله، إضافة إلى زاوية الاستدارة. بالنسبة للمحور كان Vector3.up أي المحور المتجه من أسفل لأعلى مما يجعل الاستدارة أفقية حول المحور العمودي. أما زاوية الاستدارة فهي حاصل ضرب ثلاث قيم: الأولى هي mouseDelta.x والتي تمثل مقدار الإزاحة الأفقية للفأرة منذ الإطار السابق. هذه القيمة تكون موجبة إذا تحركت الفأرة من اليسار لليمين، مما يعطينا دورانا في اتجاه عقارب الساعة كما في الجزء (ب) من الشكل 22، ودورانا بعكس عقارب الساعة لو تحركت الفأرة في الاتجاه الآخر. القيمة الثانية horizontalMouseSpeed تمثل سرعة استدارة الكائن (الأسطوانة) أفقيا. في معظم الألعاب تكون هذه القيمة قابلة للتغيير من قبل اللاعب لتتناسب مع السرعة التي اعتاد عليها لتحريك الفأرة. أما القيمة الأخيرة فهي Time.deltaTime والتي اعتدنا على رؤيتها عندما يتعلق الأمر بالحركة، حيث أننا نتعامل مع سرعة يجب أن نقوم بتحويلها إلى مسافة إزاحة أو زاوية دوران.

في السطر 50 قمنا بحساب الاستدارة الحالية للكاميرا على محورها المحلي x، وهي قيمة محصورة بين 0 و 360، وتزيد بدوران الكاميرا باتجاه عقارب الساعة (أي أنها تزيد إذا نظرت الكاميرا نحو الأسفل) كما في الجزء (ت) من الشكل 22. في الأسطر 54 إلى 56 نقوم بتحويل هذه القيمة إلى زاوية بين 180 و 180- وذلك عن طريق تحويل قياس الزاوية التي تزيد عن 180 إلى قيمة سالبة (مثلا 190 تصبح 170- وهكذا). سنقوم بتخزين هذه القيمة في المتغير currentRotation لنستفيد منها لاحقا لحساب الحدود الدوران العمودي المسموح بها للكاميرا.

بعد ذلك في السطرين 59 و 60 نقوم بحساب قيمة الدوران التي سننفذها خلال الإطار الحالي وهي تحسب باستخدام القيم الثلاث mouseDelta.y- و verticalMouseSpeed و Time.deltaTime بطريقة مشابهة لحساب الدوران الأفقي باستثناء استخدامنا للقيمة السالبة للإزاحة العمودية الفأرة وهي mouseDelta.y-. السبب في ذلك هو أن الإزاحة من أسفل لأعلى تعطينا قيمة موجبة للمتغير mouseDelta.y، وهي التي نريد تحويلها لاستدارة نحو الأعلى (بعكس عقارب الساعة) مما يجعلنا في حاجة إلى قيمة سالبة، والعكس صحيح عند الإزاحة من أعلى لأسفل. بعد حساب قيمة الدوران نقوم بتخزينها في المتغير ang.

بعد أن قمنا بحساب كمية الدوران المطلوبة، بقي علينا إتمام الخطوة الثانية بتدوير الكاميرا حول محورها الأفقي. عند حساب قيمة ang لدينا ثلاث احتمالات: الاحتمال الأول أن الفأرة لم تتحرك عموديا على الإطلاق وبالتالي تكون قيمة ang تساوي صفرا، بالتالي لا نحتاج لفعل أي شيء. الاحتمال الثاني أن تكون الفأرة تحركت نحو الأعلى، أي أن قيمة ang سالبة وتعطينا دورانا للأعلى بعكس عقارب الساعة. هذه الحالة نقوم بفحصها في السطر 65 للتأكد من أن الزاوية الجديدة نحو الأعلى، والتي ستنتج من جمع ang مع currentRotation ستكون أكبر من الحد الأدنى المسموح به وهو maxVerticalAngle-. الاحتمال الثالث والأخير هو أن تكون الفأرة تحركت نحو الأسفل، مما سيعطينا قيمة موجبة للمتغير ang بالتالي علينا أن نتأكد من أن جمع قيمتها مع زاوية الدوران الحالية currentRotation لن تزيد عن القيمة القصوى المسموح بها وهي maxVerticalAngle وهذا ما نفعله في السطر 66. في حال تحقق أحد هذين الشرطين فإننا نقوم بتنفيذ الدوران حول المحور x المحلي للكاميرا بنفس المقدار ang الذي سبق وحسبناه، هذا الدوران يتم عن طريق استدعاء ()transform.RotateAround وذلك في السطر 67.

في الأسطر من 74 إلى 92 نقوم بفحص مدخلات الجهات الأربع وإضافة الاتجاه المناسب إلى السرعة النهائية: أماما أو خلفا، يمينا أو يسارا. باقي الأسطر لن أتطرق لشرحها لأنها تؤدي نفس الوظائف التي سبق وتم شرحها في نظام إدخال ألعاب المنصات، يمكنك العودة للشرح في الصفحة 30. يمكنك أيضا مشاهدة النتيجة النهائية في المشهد scene5 في المشروع المرفق.

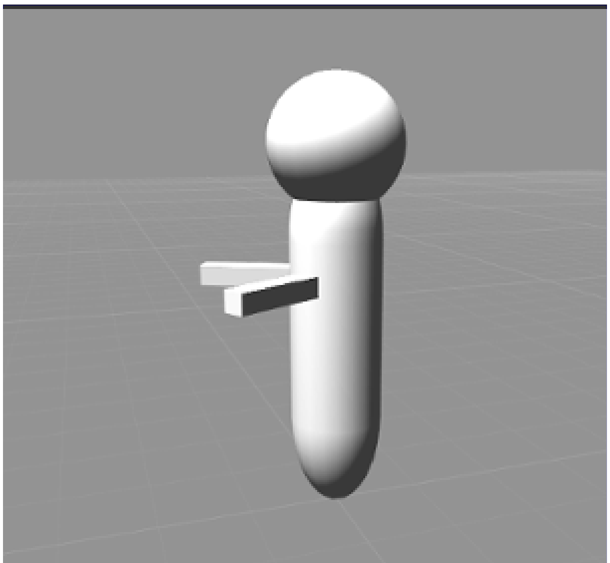
الفصل الخامس: تطبيق نظام إدخال ألعاب منظور الشخص الثالث

تضم ألعاب منظور الشخص الثالث مجموعة من أشهر سلاسل الألعاب مثل Tomb Rider و Hitman و

Splinter Cell و Max Payne وغيرها الكثير. يعتمد نظام التحكم في هذه الألعاب على رؤية شخصية اللاعب من الخلف بمسافة وارتفاع محددين يمكن أن يتغيرا خلال اللعب حسب الحالة، مثل تقريب الكاميرا إلى السلاح الذي تحمله شخصية اللاعب عند التصويب وإبعادها عند الحركة بسرعة عالية.

سنتعلم في هذا الفصل كيفية بناء هذا النوع من نظام التحكم، مستفيدين من طرق إدخال لوحة المفاتيح والفأرة، إضافة إلى العلاقات بين الكائنات ووظائف الدوران مثل ()transform.RotateAround وكلها سبق وتعرفنا عليها في فصول سابقة. هناك تقنيات متعددة تتعلق بالعلاقة بين دوران الكاميرا وحركة الشخصية داخل اللعبة. وهذه التقنيات تختلف من لعبة لأخرى ومن نوع لآخر. على سبيل المثال، في لعبة مثل World of Warcraft يكون الاعتماد بشكل أساسي على مؤشر الفأرة للتعامل مع البيئة المحيطة بالتالي فإن دوران الكاميرا يعتمد على الحركة بالأسهم، بالإضافة لذلك، يمكن استخدام زر الفأرة الأيمن لإدارة الكاميرا. أما في ألعاب أخرى مثل Hitman فإن مؤشر الفأرة يكون أداة للتصويب، لذا فإن نظر الشخصية يتوجه دائما لموقع مؤشر الفأرة وتتم عملية التصويب في ذلك الاتجاه، في هذه الحالة تدور الكاميرا تلقائيا لتتبع اتجاه النظر.

سنتعامل في هذا الفصل مع نظام بسيط يقوم بإدارة شخصية اللاعب بناء على حركة الفأرة الأفقية (كما في الفصل السابق عندما تعاملنا مع نظام منظور الشخص الأول) بالإضافة إلى تحويل الحركة العمودية للفأرة إلى حركة للكاميرا نحو الأعلى والأسفل. المهم هو أن تبقى الكاميرا في كل هذه الأحوال تنظر إلى شخصية اللاعب، وتتعقبه أينما ذهب، وهذا الأمر سنطبقه عن طريق العلاقة بين كائن الكاميرا وكائن شخصية اللاعب. أخيرا، سنسمح للاعب بتقريب الكاميرا وإبعادها عن شخصية اللاعب مستخدما عجلة الفأرة. لنقم في البداية ببناء شخصية بسيطة مستخدمين الأشكال الأساسية كما في الشكل 23.



الشكل 23: شخصية بسيطة لاستخدامها في نظام تحكم منظور الشخص الثالث

بعد أن نقوم ببناء الشخصية مستخدمين الجسم الرئيسي (وهو كائن من نوع كبسولة Capsule) كأب لجميع الكائنات الأخرى (الرأس واليدين) علينا أن نقوم أيضا بإضافة كائن الكاميرا كابن لهذا الجسم، وذلك حتى تتحرك الكاميرا خلف شخصية اللاعب دائما. الخطوة التالية هي إضافة بريمج على كائن اللاعب من أجل التحكم في حركته، إضافة إلى بريمج آخر على كائن الكاميرا. لنبدأ بالبريمج الأول ThirdPersonControl وهو الذي سنضيفه على كائن شخصية اللاعب (الكبسولة) ومهمته الاستجابة لمدخلات لوحة المفاتيح التي تحرك الشخصية في الاتجاهات الأربع إضافة إلى القفز كما هو موضح في السرد 10.

```
using UnityEngine;
using System.Collections;

public class ThirdPersonControl : MonoBehaviour {

      //السرعة العمودية للاعب عند بداية القفز
      public float jumpSpeed = 1;

      //سرعة السقوط
      public float gravity = 3;

      //سرعة الحركة الأفقية على الأرض
      public float movementSpeed = 5;

      //تخزين سرعة اللاعب من أجل تنفيذ الحركة
      private Vector3 speed;

      void Start () {

      }

      void Update () {

            //تحديث الحركة
            if(Input.GetKey(KeyCode.A)){
                  //لليسار
                  speed.x = -movementSpeed * Time.deltaTime;
            } else if(Input.GetKey(KeyCode.D)){
                  //لليمين
                  speed.x = movementSpeed * Time.deltaTime;
            } else {
                  speed.x = 0;
            }

            if(Input.GetKey(KeyCode.W)){
                  //للأمام
                  speed.z = movementSpeed * Time.deltaTime;
            } else if(Input.GetKey(KeyCode.S)){
                  //للخلف
                  speed.z = -movementSpeed * Time.deltaTime;
            } else {
                  speed.z = 0;
```

```
			}

			//قراءة مدخل القفز
			if(Input.GetKeyDown(KeyCode.Space)){
				//تنفيذ القفز فقط في حال كون الشخصية تقف على الأرض//
				if(transform.position.y == 2.0f){
					speed.y = jumpSpeed;
				}
			}

			//تحريك الشخصية
			transform.Translate(speed);

			//تطبيق الجاذبية  في حالة القفز
			if(transform.position.y > 2.0f){
				speed.y = speed.y - gravity * Time.deltaTime;
			} else {
				speed.y = 0;
				Vector3 newPosition = transform.position;
				newPosition.y = 2.0f;
				transform.position = newPosition;
			}

		}
}
```

السرد 10: قراءة مدخلات الحركة لنظام تحكم منظور الشخص الثالث

جميع أن هذه الوظائف سبق وتطرقنا لها في الفصلين السابقين بالشرح المفصل تحديدا في السرد 6 في الصفحة 30 والسرد 9 في الصفحة 44. لاحظ أن طريقة التحكم مشابهة تماما لطريقة تحكم نظام منظور الشخص الأول، باستثناء ما يتعلق بقراءة مدخلات الفأرة وتحريك الكاميرا، وهي التي سنجدها في البريمج الثاني ThirdPersonCamera والذي سنضيفه على كائن الكاميرا. هذا البريمج موضح في السرد 11.

```
using UnityEngine;
using System.Collections;

public class ThirdPersonCamera : MonoBehaviour {

	//سرعة الدوران الأفقي لشخصية اللاعب//
	public float horizontalSpeed = 0.4f;

	//سرعة الحركة العمودية للكاميرا//
	public float verticalSpeed = 5;

	//القيمة القصوى العليا والدنيا المسموح بها//
	//لارتفاع الكاميرا//
	public float minCameraHeight = 0.25f;
	public float maxCameraHeight = 15;

	//متغيرات التحكم بالتقريب//
```

```
    public float maxZoom = -10;
    public float minZoom = -30;
    public float zoomSpeed = 3;

    //هل يجب أن تتحرك الكاميرا للأسفل
    //عند تحريك الفأرة للأعلى؟ أي عكس الحركة العمودية
    public bool invertYMovement = true;

    //موقع الفأرة خلال الإطار السابق
    //ضروري لحساب كمية الإزاحة
    private Vector3 lastMousePosition;

    //متغير لتخزين كائن شخصية اللاعب
    Transform playerBody;

    void Start () {
          lastMousePosition = Input.mousePosition;
          //من الضروري أن يكون كائن اللاعب أبا لكائن الكاميرا
          playerBody = transform.parent;
    }

    void Update () {
          Vector3 mouseDelta = Input.mousePosition - lastMousePosition;

          //الإزاحة  الأفقية للفأرة يتم تحويلها إلى
          //دوران أفقي لكائن اللاعب
          playerBody.RotateAround(
                      Vector3.up, //محور الدوران
                      mouseDelta.x *
                      horizontalSpeed *
                      Time.deltaTime);//زاوية الدوران

          //الإزاحة العمودية للفأرة تتم ترجمتها إلى
          //إزاحة عمودية للكاميرا
          float yDelta = 0;
          if(invertYMovement){
               //عكس اتجاه الحركة العمودية: الفأرة للأعلى = الكاميرا للأسفل
               yDelta = -mouseDelta.y * verticalSpeed * Time.deltaTime;
          } else {
               //استخدام الاتجاه الأصلي للحركة العمودية: الفأرة للأعلى = الكاميرا للأعلى
                yDelta = mouseDelta.y * verticalSpeed * Time.deltaTime;
          }

          //تنفيذ الحركة العمودية للكاميرا
          transform.Translate(0, yDelta, 0, Space.World);

          //فحص إذا ما تجاوزت الكاميرا حدود الحركة المسموح بها عموديا
          Vector3 newCameraPos = transform.localPosition;
          if(newCameraPos.y > maxCameraHeight){
                newCameraPos.y = maxCameraHeight;
          } else if(newCameraPos.y < minCameraHeight){
                newCameraPos.y = minCameraHeight;
          }

```

```
72.            //تغيير موقع الكاميرا للموقع الجديد بعد تصحيح الموقع العمودي
73.            transform.localPosition = newCameraPos;
74.
75.            //إبقاء الكاميرا تنظر إلى كائن شخصية اللاعب
76.            transform.LookAt(playerBody);
77.
78.            //تخزين موقع الفأرة الحالي استعدادا للإطار المقبل
79.            lastMousePosition = Input.mousePosition;
80.
81.            //تنفيذ التقريب
82.            float wheel = Input.GetAxis("Mouse ScrollWheel");
83.            //تقريب الكاميرا
84.            if(wheel > 0 && transform.localPosition.z < maxZoom){
85.                //Z تحريك الكاميرا للأمام على محورها المحلي
86.                //zoomSpeed باستخدام متغير السرعة
87.                transform.Translate(0, 0, zoomSpeed);
88.            } else //إبعاد الكاميرا
89.            if(wheel < 0 && transform.localPosition.z > minZoom){
90.                //z تحريك الكاميرا للخلف على محورها المحلي
91.                //zoomSpeed باستخدام القيمة السالبة للمتغير
92.                transform.Translate(0, 0, -zoomSpeed);
93.            }
94.        }
95. }
```

السرد 11: آلية تحريك الكاميرا في نظام تحكم منظور الشخص الثالث

لاحظ أن هذا البريمج وبالرغم من كونه مضافا على كائن الكاميرا، إلا أن له تأثيرا على كل من كائن الكاميرا وكائن شخصية اللاعب كما سنرى. في البداية قمنا بتعريف بعض المتغيرات التي ستساعدنا على التحكم بحركة الكاميرا، وذلك في الأسطر 7 إلى 24. تساعدنا هذه المتغيرات على التحكم بسرعة الدوران الأفقي للاعب والحركة العمودية للكاميرا نحو الأعلى والأسفل، بالإضافة إلى حدود الحركة العمودية ومدى وسرعة تقريب أو إبعاد الكاميرا عن كائن اللاعب. لاحظ في السطرين 18 و 19 أن القيم العليا والدنيا لتقريب وإبعاد الكاميرا هي قيم سالبة؛ ذلك لأن الكاميرا سواء كانت قريبة أو بعيدة لا بد وأن تكون خلف اللاعب أي نحو الخارج، وهو الاتجاه السالب للمحور z حسب قاعدة اليد اليسرى التي نتبعها.

في الدّالّة ()Start وتحديدا في السطر 36 قمنا باستخدام المتغير playerBody لتخزين مكوّن Transform الخاص بكائن شخصية اللاعب، وتمكنّا من الوصول إليه عبر transform.parent والتي تعطينا الأب الحالي للكائن. وبما أننا قمنا بإضافة البريمج على كائن الكاميرا الذي هو بالأصل ابن لكائن شخصية اللاعب، فإن transform.parent ستوصلنا إلى كائن اللاعب.

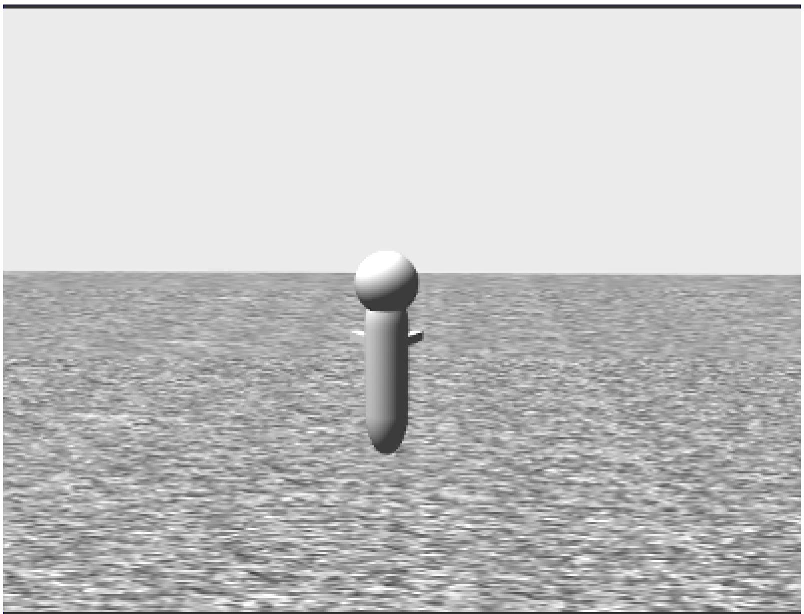
في الأسطر من 44 إلى 45 نقوم بتحويل الإزاحة الأفقية للفأرة إلى دوران لكائن اللاعب playerBody على المحور العمودي y، بطريقة شبيهة بما قمنا به عند بناء نظام تحكم منظور الشخص الأول. في الأسطر 52 إلى 62 قمنا بتعريف المتغير yDelta والذي سنستخدمه لحساب الحركة العمودية للفأرة

بناء على قيمة المتغير invertYMovement، والذي تحدد قيمته ما إذا كانت إزاحة الكاميرا العمودية ستكون في نفس اتجاه إزاحة الفأرة أم في الاتجاه المعاكس لها، وبعد ذلك نقوم بتنفيذ الحركة العمودية للكاميرا على محاور فضاء المشهد، وذلك في السطر 62.

بعد تحريك الكاميرا عموديا نقوم في الأسطر 65 إلى 70 بالتحقق من كون الموقع العمودي للكاميرا داخل حدود القيم القصوى العليا والدنيا المسموح بها، والتي قمنا بتحديدها عن طريق المتغيرين maxCameraHeight و minCameraHeight. لاحظ هنا أننا فحصنا قيمة transform.localPosition بدلا من transform.position لتحديد الموقع. الهدف من ذلك هو جعل حدود الحركة هذه نسبية إلى موقع اللاعب، وليس إلى فضاء المشهد أو سطح الأرض؛ وذلك حتى تكون حساباتنا صحيحة في كل الأحوال حتى لو تغير الموقع العمودي للاعب كما في حالة القفز مثلا. نقوم بتخزين هذه القيمة في المتغير newCameraPos ومن ثم نقوم بفحص القيمة العليا والدنيا للعضو y حتى نتأكد من بقاء قيمته ضمن الحدود المطلوبة ونعدلها إذا لزم ذلك، وأخيرا نقوم بتخزين القيمة الجديدة التي تم تعديلها في transform.localPosition كما في السطر 73.

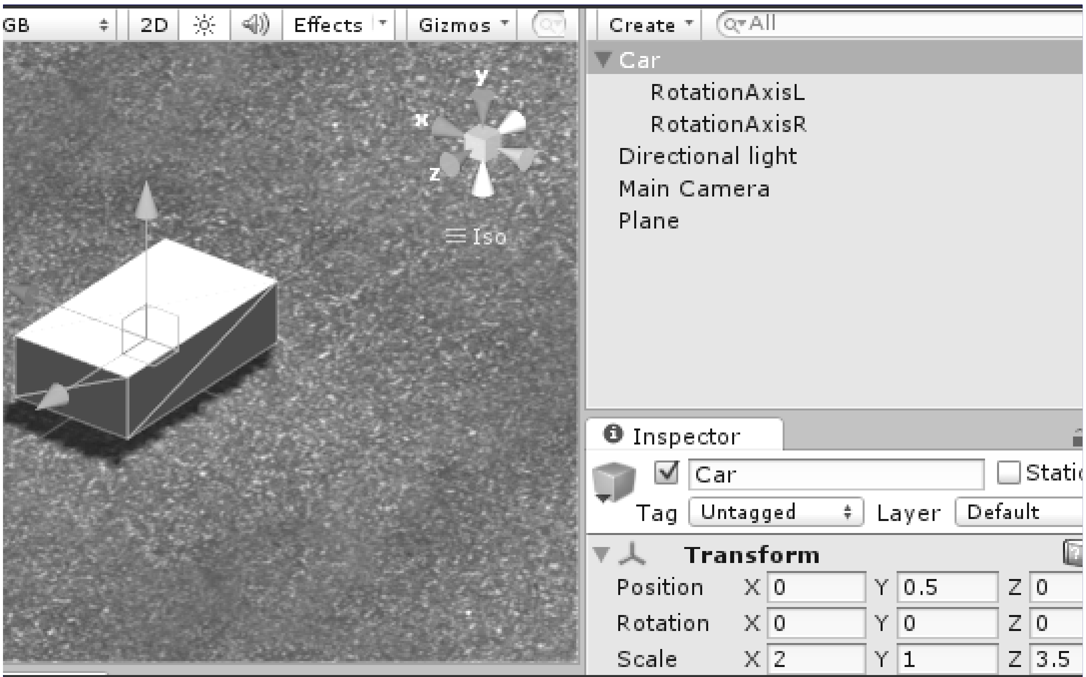
بعد الانتهاء من تحريك الكاميرا يبقى أن نتأكد من أنها تنظر دائما إلى كائن اللاعب، لذا نقوم في السطر 76 باستدعاء ()transform.LookAt ونمرر لها المتغير playerBody والذي يمثل كائن اللاعب كما ذكرنا. أخيرا في الأسطر 81 إلى 93 نقوم بقراءة مدخل عجلة الفأرة، ونترجم حركة العجلة للأعلى إلى تقريب للكاميرا نحو كائن شخصية اللاعب، وحركتها للأسفل إلى إبعاد للكاميرا. حركة الكاميرا نحو الأمام أو الخلف محكومة بالسرعة zoomSpeed ، كما أنها مشروطة بعدم تجاوزها الحدود المعرفة في maxZoom و minZoom وهي تمثل الموقعين الأقرب والأبعد المسموح بهما. مرة أخرى قمنا هنا باستخدام transform.localPosition؛ وذلك لأن القرب أو البعد هو نسبي لكائن شخصية اللاعب وليس لنقطة الأصل في المشهد.

يمكنك بناء مشهد بسيط كما في الشكل 24 لتقوم بتجربة نظام التحكم الذي قمت ببنائه، كما يمكنك الاطلاع على المشهد scene6 في المشروع المرفق لمشاهدة النتيجة النهائية.


الشكل 24: المشهد الخاص بتجربة نظام تحكم منظور الشخص الثالث

الفصل السادس: تطبيق نظام إدخال ألعاب سباق السيارات

أنظمة الإدخال الخاصة بألعاب سباق السيارات شبيهة إلى حد بعيد بأنظمة تحكم منظور الشخص الثالث من ناحية موقع الكاميرا في المشهد، مع عدم إغفال بعض الاختلافات خاصة في طريقة الحركة ودوران الكاميرا كما سنرى حالا. لتكن البداية مع مجسم يمثل السيارة، وسأستخدم هنا كائن مكعب بأبعاد (2, 1, 3.5) ليمثل السيارة. ولنقم أيضا بإضافة أرضية بمساحة كافية وهي عبارة عن كائن لوح بطول وعرض يساوي 100 (تذكر أن المساحة الافتراضية للوح هي 10 * 10، لذا فإن المحصلة النهائية عند الضرب ب 100 ستكون 1000 * 1000 أي كيلومترا مربعا واحدا). وسأستعمل إكساءً يمثل أسفلت الشارع إضافة إلى ضوء اتجاهي للمشهد. أخيرا سنقوم بإضافة كائنين فارغين كأبناء لمكعب السيارة، وذلك لاستخدامهما كمحاور للدوران. المحور الأول هو RotationAxisR وموقعه المحلي بالنسبة لمكعب السيارة هو (0 ,0 , 1) أي يبعد مترا واحدا إلى يمين السيارة، والآخر هو RotationAxisL ويبعد مترا واحدا إلى يسار السيارة أي في الموقع المحلي (0 ,0 ,1-). الشكل 25 يوضح المشهد الذي سنتعامل معه لبناء نظام تحكم ألعاب السيارات.



الشكل 25: المشهد المستخدم لتطبيق نظام تحكم ألعاب سباق السيارات

الأمر المختلف هذه المرة هو أن الكاميرا لم تتم إضافتها كابن لكائن السيارة كما كان الحال في أنظمة تحكم منظور الشخص الثالث وذلك لأسباب تتعلق بالمؤثرات كما سنرى بعد قليل. اختلاف آخر هو الطريقة التي تتحرك بها السيارة والتي تختلف عن حركة الأشخاص كما في الفصول السابقة. فالسرعة هنا ليست ثابتة وإنما تزيد بشكل تدريجي عند الانطلاق والضغط على دواسة الوقود، بينما تبدأ في التناقص عند رفع الضغط عنها. كما أن استخدام المكابح يعمل على تقليل سرعة السيارة في زمن قصير. إضافة لذلك، فإن دوران السيارة يختلف عن دوران الشخص، حيث أن هذا الأخير يدور حول نفسه دون الحاجة لمساحة تدوير. بينما السيارة تحتاج لمساحة تدور فيها، بما يشبه وجود محور افتراضي إلى اليمين أو اليسار يبعد عن السيارة مسافة معينة. هذه المسافة تقل كلما زدنا دوران عجلة القيادة مما يسمح لنا بالتحكم بزاوية الانعطاف. على الرغم من ذلك، سأقوم هنا باستخدام مسافة ثابتة لجعل الشرح أسهل، وأيضا لأننا نتعامل مع لوحة مفاتيح وليس مع نظام إدخال تناظري analog.

لنقم الآن بكتابة البريمج الذي سيسمح لنا بتحويل هذا المكعب الساكن إلى سيارة (ليس بالشكل طبعا لكن بالحركة!) يمكننا أن نتحكم بها بزيادة وإنقاص السرعة والانعطاف يمينا ويسارا. السرد 12 يوضح البريمج CarController والذي سنضيفه إلى المكعب الذي يمثل السيارة.

```
using UnityEngine;
using System.Collections;

public class CarController : MonoBehaviour {

```

```
    //السرعة القصوى للسيارة كم \ ساعة
    public float maxSpeed = 200;

    //كمية الزيادة في السرعة كم \ ساعة
    public float acceleration = 20;

    //كمية التناقص في السرعة كم \ ساعة
    public float deceleration = 16;

    //كمية التناقص في السرعة أثناء استخدام المكابح كم \ ساعة
    public float braking = 60;

    //كمية التناقص في السرعة أثناء الانعطاف نحو اليمين واليسار كم \ ساعة
    public float turnDeceleration = 30;

    //سرعة الدوران أثناء الانعطاف نحو اليمين واليسار درجة \ ثانية
    public float steeringSpeed = 70;

    //متغيرات لحفظ محوري الدوران الأيمن والأيسر
    Transform rightAxis, leftAxis;

    //سرعة السيارة الحالية متر \ ثانية
    float currentSpeed = 0;

    //قم بالضرب بهذه القيمة لتحويل السرعة من
    // كم \ ساعة إلى متر \ ثانية
    const float CONVERSION_FACTOR = 0.2778f;

    void Start () {
        //إيجاد محوري الدوران الأيمن والأيسر في قائمة الأبناء
        rightAxis = transform.FindChild("RotationAxisR");
        leftAxis = transform.FindChild("RotationAxisL");
    }

    void Update () {

        if(Input.GetKey(KeyCode.UpArrow)){
            //اللاعب يضغط على زر التسريع
            //قم بزيادة السرعة بالاعتماد على متغير التسارع
            AdjustSpeed(
               acceleration * CONVERSION_FACTOR * Time.deltaTime);
        } else {
            //اللاعب لا يضغط على زر التسريع
            //قم بتخفيض السرعة بالاعتماد على متغير تناقص السرعة
            AdjustSpeed(
              -deceleration * CONVERSION_FACTOR * Time.deltaTime);
        }

        if(Input.GetKey(KeyCode.DownArrow)){
            //Braking pressed
            //Decrease speed by braking amount
            AdjustSpeed(
                -braking * CONVERSION_FACTOR * Time.deltaTime);
        }
```

```
60.
61.            //الانعطاف نحو اليمين، وينفذ فقط في حال كانت سرعة السيارة أكبر من 5 كم \ ساعة
62.            if(Input.GetKey(KeyCode.RightArrow) &&
63.                currentSpeed > 5 * CONVERSION_FACTOR){
64.                AdjustSteering(steeringSpeed, rightAxis.position);
65.            }
66.
67.            //الانعطاف نحو اليسار، وينفذ فقط في حال كانت سرعة السيارة أكبر من 5 كم \ ساعة
68.            if(Input.GetKey(KeyCode.LeftArrow) &&
69.                currentSpeed > 5 * CONVERSION_FACTOR){
70.                AdjustSteering(-steeringSpeed, leftAxis.position);
71.            }
72.
73.            //z قم بتنفيذ الحركة للأمام على المحور المحلي
74.            transform.Translate(0, 0, currentSpeed * Time.deltaTime);
75.
76.        }
77.
78.        //الدّالّة الخاصة بإضافة قيمة جديدة إلى السرعة الحالية
79.        //تقوم بفحص الحد الأعلى للسرعة بالإضافة إلى التأكد من منع القيمة السالبة
80.        //القيمة المضافة يجب أن تكون بوحدة متر \ ثانية
81.        void AdjustSpeed(float newValue){
82.            currentSpeed += newValue;
83.            if(currentSpeed > maxSpeed * CONVERSION_FACTOR){
84.                currentSpeed = maxSpeed * CONVERSION_FACTOR;
85.            }
86.
87.            if(currentSpeed < 0){
88.                currentSpeed = 0;
89.            }
90.        }
91.
92.        //الدّالّة الخاصة بتدوير السيارة بشكل أفقي حول النقطة المزودة
93.        // بناء على السرعة المزودة بوحدة درجة \ ثانية
94.        void AdjustSteering(float speed, Vector3 rotationAxis){
95.            //تنفيذ الدوران//
96.            transform.RotateAround(
97.                rotationAxis, Vector3.up, speed * Time.deltaTime);
98.            //في حال كون السرعة أكبر من 30 كم \ ساعة
99.            //نقوم بإنقاص السرعة بمقدار متغير تناقص الانعطاف
100.           if(currentSpeed > 30 * CONVERSION_FACTOR){
101.               AdjustSpeed(
102.                   -turnDeceleration * CONVERSION_FACTOR *
103.                   Time.deltaTime);
104.           }
105.       }
106.   }
```

السرد 12: بريمج التحكم بالسيارة

في الأسطر 6 إلى 19 نقوم بتعريف عدد من المتغيرات الخاصة بسرعة حركة السيارة وهي maxSpeed والتي تمثل أقصى سرعة يمكن أن تصلها السيارة، و acceleration وهي مقدار الزيادة في السرعة مع مرور الوقت، و deceleration وهي مقدار التناقص في السرعة مع مرور الوقت، و braking والتي تمثل

التناقص في السرعة مع مرور الوقت أثناء ضغط المكابح، وأخيرا turnDeceleration والتي تمثل مقدار التناقص في السرعة مع مرور الوقت أثناء الانعطاف نحو اليمين أو اليسار. جميع هذه القيم ممثلة بوحدة كم \ ساعة كونها أسهل للتعامل معها بالنسبة لنا.

في السطر 22 قمنا بتعريف المتغير steeringSpeed والممثل بوحدة درجة \ ثانية. المتغيران rightAxis و leftAxis في السطر 25 سنستخدمهما للوصول إلى الكائنين RotationAxisR و RotationAxisL الذين سبق وأضفناهما كأبناء للسيارة على يمينها ويسارها لنقوم بإدارة جسم السيارة حولهما أثناء الانعطاف؛ ففي الواقع تحتاج السيارة إلى مساحة للانعطاف ولا تدور في مكانها، لذا نستخدم هذين الكائنين لتحديد هذه المساحة، بحيث تزيد المساحة اللازمة كلما أبعدنا الكائنين عن جسم السيارة.

في السطر 28 قمنا بتعريف متغير خاص بتخزين السرعة الحالية للسيارة والتي سنقوم بحسابها بناء على مدخلات اللاعب من تسرع ومكابح وانعطاف. لاحظ أن هذا المتغير خلافا لمتغيرات السرعة الأخرى يستخدم وحدة متر \ ثانية، ذلك لأن الوقت المعتمد في Unity هو الثانية، والوحدة الواحدة تساوي مترا واحدا مما يجعل هذه الوحدة أسهل للاستخدام مع Unity من وحدة كم \ ساعة. للتحويل من كم \ ساعة إلى متر \ ثانية قمنا في السطر 32 بتعريف متغير ذو قيمة ثابتة وهو CONVERSION_FACTOR وقيمته الثابتة هي 0.2778، وكل ما علينا هو أن نضرب أي قيمة ممثلة بـ كم \ ساعة بهذا المتغير لتحويلها إلى متر \ ثانية. بعد ذلك قمنا في الدّالّة ()Start بالبحث عن الكائنين RotationAxisR و RotationAxisL وتخزينهما في المتغيرين rightAxis و leftAxis من أجل استخدامهما لاحقا عند تطبيق الانعطاف.

قبل الشروع في شرح محتويات الدّالّة ()Update سأنتقل للأسطر 81 إلى 90، حيث قمنا هذه المرة بتعريف دالّة خاصة بنا للاستفادة منها وهي ()AdjustSpeed. إن لم تكن لك خبرة كبيرة بالبرمجة، فيمكن القول ببساطة أن تعريف الدّوال مفيد عندما تحتاج لتكرار عملية معينة عدّة مرات لكن بقيم مختلفة، وهذه العملية تتطلب منك كتابة عدّة أسطر مما يجعل أمر تكرار كتابتها متعبا. هذا تحديدا هو الحال مع عملية تغيير سرعة السيارة: حيث علينا أولا أن نضيف القيمة الجديدة إلى السرعة الحالية، ومن ثم نتأكد من أن السرعة الجديدة أقل من السرعة القصوى وتعديلها إن كانت أكبر، كما أن علينا أيضا أن نتأكد أن السرعة ليست أقل من صفر، وتغييرها إلى صفر إن كانت أقل. كل ما علينا الآن هو استدعاء هذه الدّالّة في كل مرة نرغب فيها بتغيير السرعة، وتزويدها بالقيمة newValue التي نرغب بإضافتها أو طرحها من السرعة. وقبل ذلك علينا أن نحولها للوحدة المناسبة، كون الدّالّة ()AdjustSpeed تتعامل مع الوحدة متر \ ثانية.

قمنا أيضا بتعريف دالّة أخرى في الأسطر من 94 إلى 105 وهي ()AdjustSteering، والتي سنستعملها للانعطاف يمينا أويسارا. عند استدعاء هذه الدّالّة علينا أن نقوم بتزويدها بالنقطة التي نريد تنفيذ الدوران حولها، بالإضافة إلى سرعة الانعطاف. عند استدعائها تقوم ()AdjustSteering بإدارة الكائن حول

النقطة التي زودناها بها وهي rotationAxis مستخدمة متغير السرعة speed. بعد ذلك تقوم الدّالّة بفحص سرعة السيارة الحالية، بحيث تقوم بإنقاصها بمقدار turnDeceleration في حال كون السرعة أكبر من 30 كم \ في الساعة. لاحظ أن ()AdjustSteering تقوم بإنقاص السرعة عن طريق استدعاء () AdjustSpeed، وهذا ممكن حيث أن لغة البرمجة تسمح لنا باستدعاء دالّة من داخل دالّة أخرى.

بالعودة الآن إلى ()Update وتحديدا الأسطر من 42 إلى 52، حيث نقوم بفحص مفتاح السهم العلوي للوحة المفاتيح، وزيادة سرعة السيارة بمقدار التسارع acceleration في حال كان اللاعب يضغط على هذا المفتاح. لاحظ أننا قمنا باستدعاء ()AdjustSpeed حتى تقوم بكل ما يلزم بخصوص تغيير السرعة، وقمنا عند استدعائها بضرب التسارع بمتغير التحويل CONVERSION_FACTOR لتحويل قيمة التسارع إلى متر \ ثانية، حيث أن Time.deltaTime تعطينا الوقت المنقضي بالثواني كما أن () AdjustSpeed تتعامل فقط مع هذه الوحدة، واستدعاؤها دون إجراء التحويل سيعطينا قيما خاطئة. في حال كون اللاعب لا يضغط على مفتاح السهم العلوي، نقوم بإنقاص سرعة السيارة بمقدار التناقص deceleration مع مراعاة استخدام القيمة السالبة هذه المرة، كوننا نريد طرح القيمة من السرعة الحالية وليس زيادتها عليها.

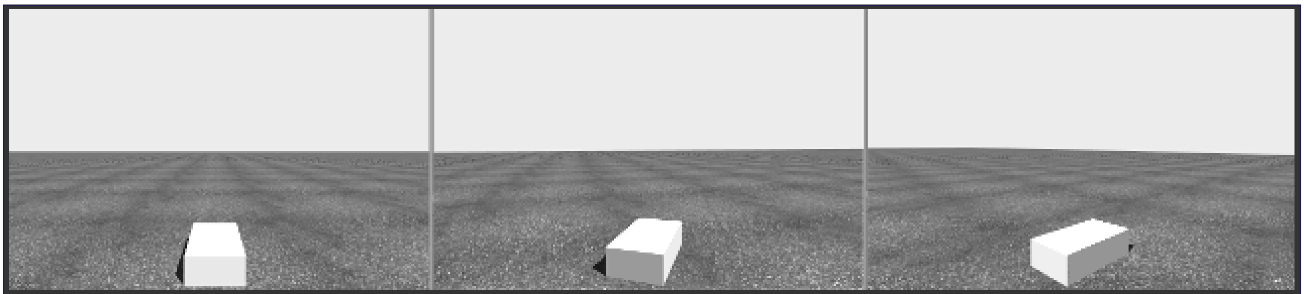
في الأسطر من 54 إلى 59 نقوم بفحص مفتاح السهم الأسفل، وتشغيل المكابح في حال كون اللاعب يضغط عليه. المكابح هنا عبارة عن إنقاص للسرعة ولكن بمقدار أكبر من deceleration وهو braking، مما يعني أن السيارة مع المكابح ستحتاج إلى وقت أقل للتوقف لأن التناقص في السرعة يكون أكبر في كل ثانية.

تنفيذ الانعطاف يتم في الأسطر 62 إلى 71، حيث نقوم بفحص حالة كل من السهم الأيمن وسرعة السيارة الحالية، حيث لا ننعطف إذا كانت سرعة السيارة تقل عن 5 كم \ في الساعة. الهدف من هذا هو ألا نسمح لمفاتيح الانعطاف بأن تحرك السيارة إذا كانت متوقفة. لاحظ أننا في السطر 64 قمنا باستخدام النقطة rightAxis كمركز للدوران مع القيمة الموجبة للمتغير steeringSpeed وذلك لأن الدوران لليمين يكون مع عقارب الساعة عندما يكون محور الدوران هو الاتجاه الموجب للمحور y (السطر 97). أما في السطر 70 فإننا نستخدم القيمة السالبة للمتغير steeringSpeed لإحداث دوران بعكس عقارب الساعة حول النقطة leftAxis.

أخيرا وبعد حساب السرعة و تنفيذ الانعطاف، بقي أن نقوم بتحريك السيارة نحو الأمام على محورها المحلي z باستخدام قيمة currentSpeed مضروبة في الوقت المنقضي. تذكر أن currentSpeed تم حسابها بوحدة متر \ ثانية لذا يمكن ضربها في Time.deltaTime مباشرة، كما في السطر 74. يمكنك الآن تجربة نظام التحكم بالسيارة كونه أصبح جاهزا. وذلك قبل الانتقال للخطوة التالية وهي كتابة البريمج الخاص بالكاميرا.

الكاميرا الخاصة بنظام تحكم ألعاب السيارات تتبع السيارة في حركتها بمسافة محددة خلفها، ولها ارتفاع

محدد عن ارتفاع السيارة. إضافة لذلك، فإن دوران الكاميرا خلف السيارة لا يكون فوريا، وإنما يتأخر قليلا ويكون بسرعة محددة تكون عادة أقل من سرعة انعطاف السيارة. نتيجة لذلك، فإن طول زمن الانعطاف للسيارة سيؤدي إلى دورانها أمام الكاميرا وظهور جزء أكبر من جانبها للاعب. هذا التأثير موضح في الشكل 26.



الشكل 26: دوران السيارة أمام الكاميرا أثناء الانعطاف وظهور جانب السيارة للاعب

البريمج الذي سنضيفه للكاميرا هو CarCamera الموضح في السرد 13.

```
using UnityEngine;
using System.Collections;

public class CarCamera : MonoBehaviour {

        //متغير لتخزين كائن السيارة
        public Transform car;

        //ارتفاع الكاميرا عموديا فوق ارتفاع السيارة
        public float height = 4;

        //بعد الكاميرا من الخلف عن السيارة
        //بغض النظر عن الارتفاع
        public float zDistance = 10;

        //عدد ثواني الانتظار قبل البدء بتدوير الكاميرا
        //بعد أن تقوم السيارة بالانعطاف
        public float turnTimeout = 0.25f;

        //سرعة دوران الكاميرا
        //بوحدة درجة \ ثانية
        public float turnSpeed = 50;

        //الوقت المنقضي منذ تغير الزاوية بين الكاميرا والسيارة
        //إلى قيمة مرتفعة
        float angleChangeTime = -1;

        void Start () {
                //يتم وضع الكاميرا مبدئيا في نفس
```

```
            //موقع السيارة وإدارتها بنفس دورانها
            transform.position = car.position;
            transform.rotation = car.rotation;
    }

    //نقوم باستخدام LateUpdate لنضمن تحرك السيارة
    //قبل تحريك الكاميرا
    void LateUpdate () {
        //نبدأ أولا بوضع الكاميرا في نفس موقع السيارة
        //في فضاء المشهد
        transform.position = car.position;

        //نقوم الآن بتحريك الكاميرا نحو الخلف
        // على محورها المحلي z
        //و لأعلى بناء على متغيري المسافة والارتفاع
        transform.Translate(0, height, -zDistance);

        //حساب الزاوية الأفقية بين اتجاه نظر الكاميرا واتجاه مقدمة السيارة
        float angle = Vector3.Angle(car.forward, transform.forward);

        //تأكد من كون قياس الزاوية أكبر من المنطقة الميتة
        //قياس الزاوية سيكون دائما موجبا بغض النظر
        //عن اتجاه انعطاف السيارة
        if(angle > 1){
            //الفرق كبير كفاية لتدوير الكاميرا
            //لنتأكد أولا إن كان الوقت قد حان للبدء بتدوير الكاميرا
            if(angleChangeTime == -1){
                angleChangeTime = 0;
            }

            //نضيف الوقت المنقضي من الإطار السابق
            //إلى مجموع الوقت منذ بداية انعطاف السيارة
            angleChangeTime += Time.deltaTime;

            if(angleChangeTime > turnTimeout){
                //حان الوقت للبدء بتدوير الكاميرا
                //نقوم أولا بعملية الضرب الاتجاهي لكل من اتجاه نظر الكاميرا
                //واتجاه مقدمة السيارة
                float resultDirection =
                    Vector3.Cross(car.forward,
                            transform.forward).y;

                //إشارة المتغير y
                //في المتجه الناتج تحدد اتجاه الدوران المطلوب
                float rotationDirection;
                if(resultDirection > 0){
                    rotationDirection = -1;
                } else {
                    rotationDirection = 1;
                }

                //نقوم الآن بإدارة الكاميرا حول السيارة مستخدمين اتجاه
                //الدوران وقيمة سرعة الدوران
```

```
83.                     transform.RotateAround(car.position,
84.                                            Vector3.up,
85.                       rotationDirection * turnSpeed *
   Time.deltaTime);
86.                 }
87.             } else {
88.                 //الفرق في الزاوية صغير جدا
89.                 //نقوم بإعادة الوقت المنقضي إلى القيمة الافتراضية
90.                 angleChangeTime = -1;
91.             }
92.         }
93. }
```

السرد 13: الكاميرا الخاصة بنظام تحكم ألعاب السيارات

تذكر أننا هنا نتعامل مع كائن كاميرا مستقل عن كائن السيارة، بخلاف ما كان عليه الحال مع كاميرا منظور الشخص الأول ومنظور الشخص الثالث. من أجل ذلك قمنا بتعريف المتغير car والذي سنقوم بربطه مع كائن السيارة كما سبق وفعلنا في نظام تحكم ألعاب المنصات (انظر الشكل 19 صفحة 34)، بالإضافة إلى كل من المتغير height الذي يحدد ارتفاع الكاميرا عن مستوى ارتفاع السيارة والمتغير zDistance لتحديد المسافة الأفقية بين السيارة والكاميرا (أي بعد الكاميرا عن السيارة من الخلف).

للتحكم في دوران الكاميرا قمنا بتعريف المتغير turnTimeout وهو الوقت الذي ستنتظره الكاميرا قبل أن تبدأ بالدوران بعد أن تنعطف السيارة. أمّا turnSpeed فتتحكم في سرعة دوران الكاميرا. أخيرا قمنا بتعريف المتغير angleChangeTime، والهدف منه هو تسجيل الوقت الذي انعطفت فيه السيارة وأصبحت قياس الزاوية بين مقدمتها وبين اتجاه نظر الكاميرا أكبر من درجة واحدة. الهدف من تسجيل هذا الوقت هو حساب الزمن المنقضي منذ تغير الزاوية، حتى نبدأ بتدوير الكاميرا خلف السيارة بعد أنقضاء الوقت المحدد بـ turnTimeout. سنتطرق لشرح أهمية هذا المتغير بالتفصيل بعد قليل.

الخطوة الأولى مع الدّالّة ()Start في السطر 28، وهي أن نتأكد من أن الكاميرا موجودة في نفس موقع السيارة، وأنها تنظر في الاتجاه الذي تشير إليه مقدمة السيارة. لذا قمنا بنسخ قيم الموقع والدوران من كائن السيارة إلى كائن الكاميرا. سألفت الانتباه هنا إلى قضية برمجية بسيطة وهي الفرق بين نسخ القيمة ونسخ المؤشر. في حال نسخ القيمة كما فعلنا هنا تبقى المتغيرات مستقلة عن بعضها البعض بعد إتمام عملية النسخ، بمعنى أن أي تغير مستقبلي على قيمة car.position أو car.rotation لن يؤثر على متغيرات الكاميرا transform.position و transform.rotation، بينما يختلف الموضوع في حالة نسخ المؤشر، لكنني سأؤجل الحديث عنها حتى الوقت المناسب لذلك.

لاحظ في السطر 37 أننا استخدمنا ()LateUpdate وليس ()Update؛ وذلك حتى نضمن قيام Unity بتنفيذ CarCotroller أولا لتحديث موقع السيارة قبل تنفيذ CarCamera لتحريك الكاميرا (يمكنك العودة لشرح ()LateUpdate في الصفحة 35). في السطرين 40 و 45 قمنا بوضع الكاميرا في المكان المناسب بالنسبة للسيارة، وذلك على خطوتين: الخطوة الأولى هي تحريك الكاميرا إلى نفس موقع

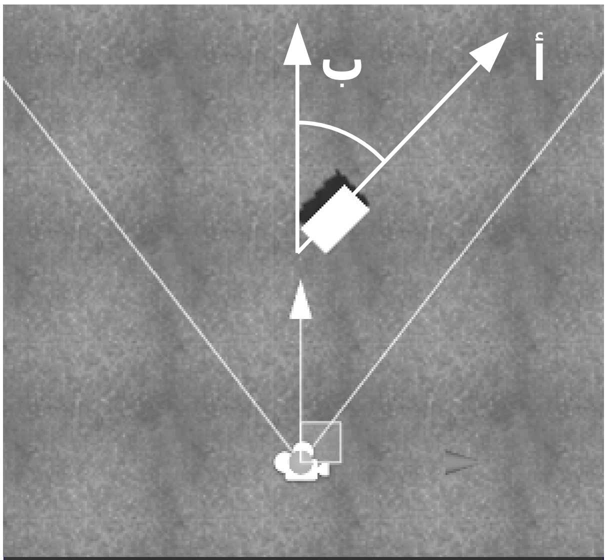
السيارة (السطر 40) ومن ثم استخدام ()transform.Translate لتحريكها نحو الخلف والأعلى على محاور فضائها المحلي وفقا للقيم height و zDistance. استخدام القيمة السالبة لـ zDistance الهدف منه أن تتحرك الكاميرا نحو الخلف وليس الأمام حسب قاعدة اليد اليسرى.

بعد تحديث موقع الكاميرا حان الوقت لتحديث الدوران. الخطوة الأولى لذلك هي أن نقوم بحساب الزاوية بين الاتجاه الذي تشير إليه مقدمة السيارة car.forward والاتجاه الذي تنظر إليه الكاميرا transform.forward، المقصود بالمتجه forward هنا هو الاتجاه الموجب لمحور الفضاء المحلي z لكل من السيارة والكاميرا. هذا الحساب قمنا بتنفيذه في السطر 48 وتخزين قيمة الزاوية في المتغير angle. من المهم الإشارة هنا إلى أن الدّالّة ()Vector3.Angle تقوم بحساب أصغر زاوية ممكنة بين المتجهين الذين نقوم بتمريرهما لها وإرجاع القيمة على شكل قياس زاوية موجب، بغض النظر عن كون المتجه الأول إلى يمين أو يسار المتجه الثاني.

جميع الخطوات اللاحقة من السطر 53 إلى السطر 85 والمتعلقة بتدوير الكاميرا تعتمد على كون قياس الزاوية angle أكبر من درجة واحدة، وهو الشرط الذي نقوم بفحصه في السطر 53 قبل الانتقال إلى الخطوات اللاحقة، والتي تبدأ بالتأكد من أن قيمة متغير الوقت المنقضي منذ تغير الزاوية angleChangeTime لا تساوي 1-، وإعادتها إلى 0 في تلك الحالة (الأسطر 56 إلى 58). بعد ذلك نقوم بإضافة الوقت المنقضي منذ الإطار السابق إلى مجموع الوقت المنقضي منذ تغير الزاوية، وذلك في السطر 62. في السطر 64 نقوم بالتأكد من أن مجموع الوقت المنقضي منذ تغير الزاوية بين السيارة والكاميرا قد تجاوز فترة الانتظار المحددة في المتغير turnTimeout، ونبدأ بتدوير الكاميرا إذا ما تحقق هذا الأمر.

لتدوير الكاميرا نحتاج لثلاث معلومات كما تعلمنا سابقا: محور الدوران، واتجاه الدوران مع أو عكس عقارب الساعة، وأخيرا سرعة الدوران. محور الدوران هنا هو المحور العمودي، وموقعه هو نفس موقع السيارة، حيث أن الكاميرا ستدور حول السيارة، أمّا السرعة فهي محددة سلفا في المتغير turnSpeed. بقي علينا الآن أن نعرف اتجاه الدوران، وهو يتحدد تبعا لاتجاه انعطاف السيارة؛ فإذا انعطفت لليمين، سندير الكاميرا مع عقارب الساعة لتصبح خلف السيارة، أما لو انعطفت لليسار فسيكون الدوران بعكس عقارب الساعة.

لحساب اتجاه الدوران، استخدمنا الضرب الاتجاهي في الأسطر 68 إلى 70. فائدة الضرب الاتجاهي هنا هو أنه ينتج لنا متجها متعامدا على مستوى المتجهين المضروبين، والاتجاه الذي يشير إليه سيعني لنا الكثير. فعند ضرب اتجاه مقدمة السيارة مع اتجاه نظر الكاميرا (كلاهما أفقي) فإن الناتج سيكون متجها عموديا إما نحو الأعلى أو نحو الأسفل، وذلك حسب قياس الزاوية الأصغر بين المتجهين. للتوضيح لنأخذ مثالا على الشكل 27 والذي يوضح الزاوية بين الكاميرا ومقدمة السيارة عند الانعطاف نحو اليمين.



الشكل 27: فرق الزاوية بين متجه مقدمة السيارة (أ) ومتجه اتجاه نظر الكاميرا (ب)

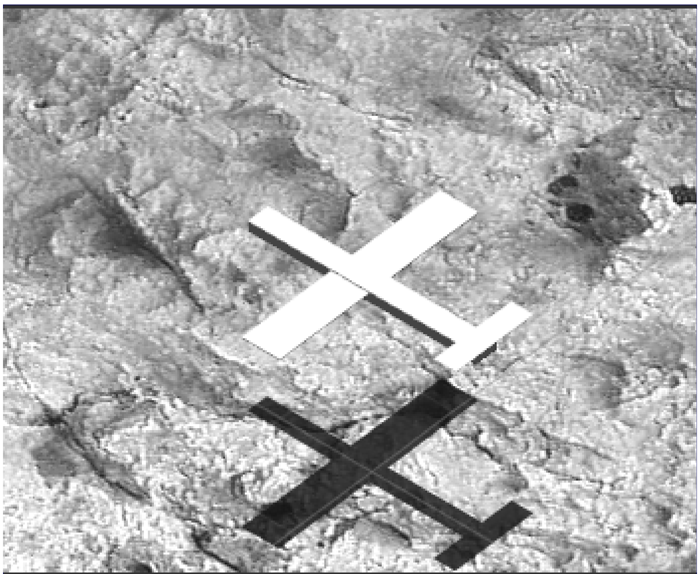
الكاميرا في الشكل 27 تنظر نحو الأمام، بينما مقدمة السيارة نحو اليمين قليلا. لمعرفة اتجاه الناتج علينا أن نستخدم قاعدة اليد اليسرى في الضرب الاتجاهي، وهي تختلف عن قاعدة اليد اليمنى المستخدمة عادة في الفيزياء والرياضيات، والتي قد تكون تعلمتها في المدرسة. حسب قاعدة اليد اليسرى، فإن أقصر مسافة يقطعها الطرف الأول في العملية (مقدمة السيارة) باتجاه الطرف الثاني (اتجاه الكاميرا) هي بعكس عقارب الساعة، مما يجعل القيمة y في المتجه الناتج سالبة (أي نحو الأسفل). وهذه القيمة ستفيدنا في معرفة اتجاه دوران الكاميرا الذي نحتاجه.

هذه القيمة نقوم بتخزينها في المتغير resultDirection ومن ثم نستفيد منها لمعرفة اتجاه الدوران rotationDirection. هذا الأمر يتم في الأسطر 74 إلى 79. فإذا كانت قيمة resultDirection سالبة، نستنتج من قاعدة اليد اليسرى أن الاتجاه الأقصر هو أن ندير السيارة بعكس عقارب الساعة، وبما أننا هنا لا نملك التحكم بالسيارة وإنما بالكاميرا، فإن اتجاه دوران الكاميرا يجب أن يكون في الاتجاه المعاكس أي مع عقارب الساعة حتى تلحق بالسيارة. لهذا السبب ترى أن إشارة rotationDirection تكون عكس إشارة resultDirection. أخيرا نقوم في الأسطر من 83 إلى 85 بتنفيذ الدوران المطلوب للكاميرا تبعا للحسابات التي قمنا بها. يمكنك مشاهدة النتيجة وتجربتها في المشهد scene7 في المشروع المرفق.

الفصل السابع: تطبيق نظام إدخال ألعاب محاكاة الطيران

آخر أنواع أنظمة التحكم التي سنتطرق لها في هذه الوحدة هي أنظمة محاكاة الطائرات. طبعا لا أقصد بالمحاكاة هنا أن أقوم ببناء نظام مشابه تماما لقمرة قيادة طائرة كالذي في Microsoft Flight Simulator، بل سأتناول نظاما بسيطا كالذي يمكن أن تراه في بعض أجزاء لعبة GTA. نوع الطائرة الذي سنتناوله سيقتصر على الطائرة النفّاثة (ذات الأجنحة) وليس الطائرة العمودية (الهيلكوبتر).

نظام تحكم الطائرات بسيط من حيث البرمجة، لأنه يعتمد على الدوران بشكل أساسي، لكنه قد يكون صعبا على اللاعب الذي لم يعتد على كيفية لعب هذه الألعاب. بما أنني هنا سأتناول حالة واحدة وهي الطيران المستمر، فلن أتطرق لأي تفاصيل تتعلق بالإقلاع أو الهبوط أو تغيير سرعة الطائرة، بل سأفترض أنها تحلق بسرعة ثابتة. لنقم في البداية ببناء نموذج طائرة بسيط كما في الشكل 28 مستخدمين مجموعة من المكعبات، ولنقم بإدراجها جميعا كأبناء لجسم الطائرة الأساسي حتى تتحرك كلها كوحدة واحدة. لنقم أيضا باستخدام أرضية كبيرة نسبيا هذه المرة، نظرا للسرعة العالية للطائرة. من أجل ذلك سأستعمل هنا أرضية بحجم 10000 * 10000 أي 10 كيلومترات مربعة، وسأستخدم إكساءا على شكل صخور وأكرره 1000 مرة على المحورين حتى تظهر الأرضية بحجم واقعي.



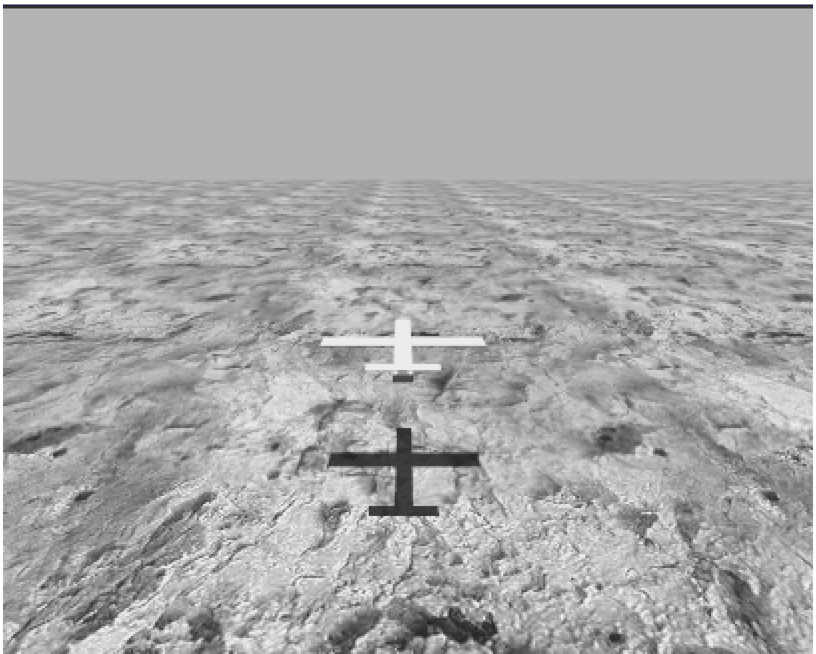
الشكل 28: نموذج طائرة بسيط مبني باستخدام الأشكال الأساسية

لننتقل الآن لكيفية التحكم بالطائرة. سندع الطائرة تطير تلقائيا باتجاه الأمام دون أي تدخل من اللاعب، وهذا أمر بديهي؛ كون الطيار لا يستطيع إيقاف الطائرة في الهواء أثناء التحليق. ما يستطيع اللاعب فعله هو توجيه الطائرة باستخدام الدوران. سنستخدم سهمي لوحة المفاتيح الأيمن والأيسر لجعل الطائرة تميل إلى اليمين أو اليسار، وذلك بتدويرها حول محور فضائها المحلي z، هذا النوع من الدوران يسمى الالتفاف(roll). النوع الآخر من الدوران هو التأرجح (pitch)، وهو الدوران حول المحور المحلي x أي نحو الأمام أو الخلف، وسنستخدم مفتاحي السهم الأعلى والأسفل لهذا النوع من التدوير.

عندما يقوم اللاعب بالضغط على مفتاح السهم الأسفل، سترتفع مقدمة الطائرة، مما يؤدي إلى ارتفاعها أكثر وأكثر عن سطح الأرض مع استمرار حركتها. العكس تماما يحدث عند الضغط على مفتاح السهم الأعلى، حيث سنقوم بخفض مقدمة الطائرة مما سيؤدي لتقليل ارتفاعها بمرور الوقت. من ناحية أخرى، فإن مفتاحي السهمين الأيمن والأيسر لن يؤثرا على ارتفاع الطائرة أو اتجاه طيرانها، وإنما يقومان

بإمالتها. فعندما تميل الطائرة نحو اليمين ومن ثم نقوم برفع مقدمتها وهي قي تلك الوضعية، فإن اتجاه طيرانها يتغير باتجاه اليمين وهكذا (تحتاج لبعض الوقت لتعتاد على الحركة إن لم تكن لديك معرفة بهذا النوع من الألعاب).

بالنسبة للكاميرا، سنقوم بوضعها خلف الطائرة بارتفاع منها قليلا، ونضيفها كابن لكائن الطائرة حتى تتبعها باستمرار كما في الشكل 29، جدير بالذكر أنه من الأفضل أن نقوم بتغيير قيمة لوح القص البعيد للكاميرا Far Clipping Plane وزيادتها إلى 5000 بدلا من 1000؛ نظرا لأن ألعاب الطائرات تعتمد على رؤية بعيدة المدى كون الطائرة تحلق على ارتفاعات شاهقة وتطير بسرعة كبيرة.أخيرا سنقوم بإضافة البريمج FlyController الموضّح في السرد 14 إلى كائن الطائرة حتى نتحكم بها.



الشكل 29: الطائرة كما تبدو من منظور الكاميرا

```
using UnityEngine;
using System.Collections;

public class FlyController : MonoBehaviour {

        //سرعة الطيران نحو الأمام مقدرة بـ متر \ ثانية
        public float flySpeed = 166;

        //z سرعة الدوران على المحور
        public float rollSpeed = 35;

        //x سرعة الدوران على المحور
        public float pitchSpeed = 35;

```

```
    void Start () {

    }

    void Update () {
        //قيم الدوران الخاصة بالإطار الحالي
        float roll, pitch;

        //z حساب الدوران حول المحور
        //اعتمادا على مفتاحي الأسهم الأيمن والأيسر
        if(Input.GetKey(KeyCode.RightArrow)){
            roll = -rollSpeed * Time.deltaTime;
        } else if(Input.GetKey(KeyCode.LeftArrow)){
            roll = rollSpeed * Time.deltaTime;
        } else {
            roll = 0;
        }

        //x حساب الدوران حول المحور
        //اعتمادا على مفتاحي الأسهم الأعلى والأسفل
        if(Input.GetKey(KeyCode.DownArrow)){
            pitch = -pitchSpeed * Time.deltaTime;
        } else if(Input.GetKey(KeyCode.UpArrow)){
            pitch = pitchSpeed * Time.deltaTime;
        } else {
            pitch = 0;
        }

        //تطبيق الدوران حول المحورين المحليين
        //z و x
        transform.Rotate(0, 0, roll);
        transform.Rotate(pitch, 0, 0);

        //flySpeed تحريك الطائرة نحو الأمام اعتمادا على قيمة السرعة
        transform.Translate(0, 0, flySpeed * Time.deltaTime);

        //منع الطائرة من الانخفاض تحت علو 5 أمتار
        Vector3 pos = transform.position;
        if(pos.y < 5){
            pos.y = 5;
        }
        transform.position = pos;
    }
}
```

السرد 14: نظام التحكم بالطائرة

كما تلاحظ فإن نظام التحكم بسيط ولا يوجد أي حسابات معقدة، كما أنه يستخدم مجموعة من الوظائف والتقنيات التي سبق شرحها. يمكنك مشاهدة النتيجة النهائية في المشهد scene8 في المشروع المرفق.

إلى هنا نصل إلى نهاية الوحدة الثانية، والتي تناولنا فيها كيفية الاستفادة من مدخلات المستخدم من

أجل السماح له بالتفاعل مع بيئة اللعبة. قمت في هذه الوحدة بشرح نوعين من أجهزة الإدخال وهما لوحة المفاتيح والفأرة، إلا أن Unity يسمح لك باستخدام أنواع أخرى من أجهزة الإدخال مثل يد التحكم وشاشات اللمس الخاصة بأجهزة الجوّال والحواسيب اللوحية. على الرغم من ذلك، فإن طريقة التعامل معها لن تختلف كثيرا عن التعامل مع لوحة المفاتيح والفأرة، حيث أنك تقوم عادة بقراءة حالات مفاتيح أو أزرار معينة بالإضافة إلى قيم إزاحة مختلفة، ومن ثم تحويلها إلى وظائف تؤثر على كائنات المشهد بطرق مختلفة.

تمارين

1. قمنا في السرد 6 (صفحة 30) ببناء نظام تحكم لألعاب المنصات. بعض ألعاب المنصات تسمح للاعب بزيادة السرعة عبر الضغط على أحد المفاتيح. قم بتعريف متغير سرعة جديد (سمّه مثلا runSpeed) بحيث تكون قيمته أكبر من السرعة العادية للحركة. ثم قم بفحص ما إذا كان اللاعب يضغط على المفتاح Shift (عن طريق فحص القيمة KeyCode.LeftShift) وزيادة السرعة عند الضغط على هذا المفتاح أو استخدام السرعة الأصلية عند عدم الضغط عليه (يمكنك تطبيق التمرين على أنظمة التحكم الأخرى إن كنت ترغب بذلك).

2. عندما قمنا بتطبيق الحركة في الأنظمة المختلفة (المنصات، منظور الشخص الأول، منظور الشخص الثالث)، لم نضع في حسابنا التسارع والتباطؤ في الحركة، والذي قد يكون مهما في حالة القفز مثلا. حاول الاستفادة من آلية التسارع والتباطؤ التي قمنا بتطبيقها في السرد 12 (صفحة 55) في تحسين أنظمة التحكم السابقة، مما يجعل اللاعب غير قادر على التوقف في الهواء عند القفز ويبقى مندفعا في الاتجاه الذي قفز إليه حتى يعاود الهبوط على الأرض. يمكنك تطبيق الأمر بأكثر من طريقة حسب ما تراه مناسبا.

3. حاول أن تستخدم بريمج الكاميرا الخاص بتتبع السيارة (السرد 13 صفحة 60) كبديل لإضافة الكاميرا كابن في نظام تحكم منظور الشخص الثالث. ما هي التغييرات التي يلزمك إجراؤها حتى تتمكن من التحكم بشخصية اللاعب بسهولة باستخدام نظام الكاميرا الجديد؟

4. حاول إجراء تعديل على نظام التحكم بالطائرة (السرد 14 صفحة 65) بحيث تقوم الطائرة بالعودة تلقائيا للوضعية الأفقية الأصلية عندما يتوقف اللاعب عن الضغط على مفاتيح التحكم. ستحتاج للقيام بحسابات شبيهة بنظام تتبع الكاميرا للسيارة؛ وذلك لتحدد اتجاه الدوران ومتى يجب أن تتوقف عنه. قم بتطبيق العودة للوضعية التلقائية على المحورين x و z.

الوحدة الثالثة: منطق اللعبة الأساسي

ألعاب الحاسب ما هي في النهاية إلا برامج كغيرها، وتخضع للمنطق الحاسوبي في التعامل مع الحقائق واتخاذ القرارات وتنفيذ المهام. سنتعرف في هذه الوحدة إن شاء الله على مجموعة من الميكانيكيات شائعة الاستخدام في الألعاب وكيفية برمجتها. هذه الميكانيكيات سبق وتعاملنا مع بعض أمثلتها البسيطة في الوحدة السابقة مثل الحركة والقفز.

بعد الانتهاء من دراسة هذه الوحدة يفترض أن تكون قادرا على:

- برمجة بعض أشكال أنظمة التصويب البسيطة

- برمجة الكائنات القابلة للجمع مثل قطع النقود والذخائر

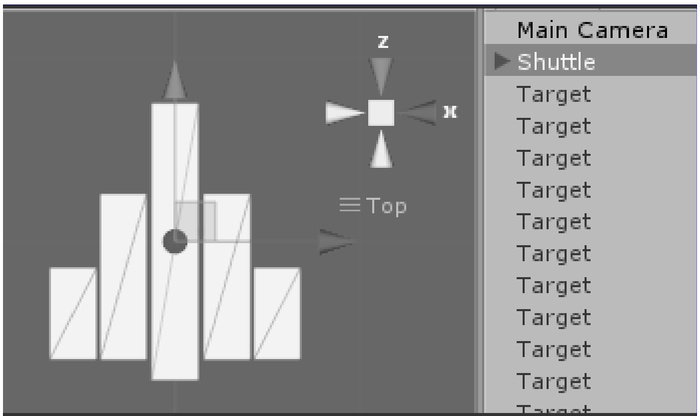
- برمجة إمساك وإفلات الكائنات المختلفة من قبل اللاعب

- برمجة المحفّزات (triggers) ونقاط التوليد (spawns) والكائنات القابلة للاستخدام

الفصل الأول: برمجة أنظمة التصويب

ميكانيكية التصويب منتشرة في الألعاب بكثرة، وذلك بغض النظر عن كون هذه الألعاب ثنائية الأبعاد أو ثلاثية الأبعاد. التصويب الذي سنتناوله في هذا الفصل يعتمد على مقذوف بسيط يسير بسرعة ثابتة نحو الأمام. لن نتناول بالتالي أية تأثيرات فيزيائية للجاذبية على المقذوف، كما لن نتطرق للمقذوفات عالية السرعة مثل رصاصة البندقية، حيث أن لها طريقة مختلفة سنتناولها في الوحدة القادمة إن شاء الله.

لنبدأ بلعبة بسيطة تشبه لعبة Space Invaders الكلاسيكية. سنقوم ببناء مركبة فضائية بسيطة كالتي في الشكل 30، ومن ثم سنقوم بكتابة بريمج للتحكم في هذه المركبة. سنكون قادرين على تحريك المركبة في أربع اتجاهات وإطلاق نوعين من المقذوفات وهي الطلقات والصواريخ، ويوجد بينهما أوجه تشابه واختلاف. لاحظ أننا سنتعامل هذه المرة مع منظور عمودي من أعلى لأسفل، لذا فإن حركة المركبة ستكون في بعدين هما x و z. بالإضافة لذلك فإننا سنقوم بتغيير إسقاط الكاميرا إلى الإسقاط العمودي حتى نرى كل المشهد في بعدين فقط فتبدوا المكعبات كأنها مستطيلات.



الشكل 30: المركبة الفضائية الخاصة بتصويب المقذوفات

ستتعرف في هذا الفصل على مفهوم جديد سيساعدنا في برمجة المقذوفات، ألا وهو القوالب prefabs. الفكرة تعتمد على إنشاء كائن وإضافة كل ما يلزمه من مكوّنات وبريمجات وإكساءات، ومن ثم تخزينه على شكل قالب. هذا القالب يمكن أن يستخدم لاحقا لإنتاج عدد غير محدد من النسخ من الكائن الأصلي، كما يمكن تغيير مجموعة كبيرة من الكائنات دفعة واحدة عن طريق تغيير القالب الذي تم استخدامه لإنتاجها. ستفيدنا القوالب في عمل الطلقات والصواريخ، حيث أن عملية التصويب تحتاج لإنتاج عدد غير محدد من الطلقات في أثناء تشغيل اللعبة.

نظرا لوجود تفاعل بين عدد من البريمجات في هذا المشهد، سأقوم بعرضها بترتيب معيّن بحيث أقوم بتأجيل البريمجات التي تعتمد في تنفيذ وظيفتها على برامج أخرى، وأبدأ بشرح البريمجات المستقلّة التي لا تعتمد على غيرها. لهذا سأبدأ ببريمج الهدف الذي ستقوم المركبة بالتصويب عليه. لبناء الهدف سنقوم باستخدام القوالب حيث سيكون لدينا عدد كبير نسبيا من الأهداف ومن الجميل أن تكون لدينا القدرة على تغييرها من مكان واحد.

لبناء قالب الهدف قم في البداية بإضافة مكعب للمشهد، ومن ثم سنضيف إلى هذا المكعب البريمج Target والذي يحتوي على متغير واحد فقط وهو hit، ومن خلاله نحدد إن كان هذا الهدف قد تمت إصابته أم لا. السرد 15 يوضح البريمج Target.

```
using UnityEngine;
using System.Collections;

public class Target : MonoBehaviour {

        //سيقوم بريمج الطلقة بتغيير هذه القيمة إلى true حالما تتم إصابة الهدف
        public bool hit = false;

        //غيّر هذه القيمة إلى true بمجرد أن تقوم باستدعاء الدّالّة Destroy()
        bool destroyed = false;
```

```
11.
12.       void Start () {
13.
14.       }
15.
16.       void Update () {
17.             if(hit){
18.                   //أصابت الطلقة الهدف، لذا قم بتشغيل حركة التدمير وهي عبارة عن دوران
19.                   //وتصغير في الحجم
20.                   transform.Rotate(0, 720 * Time.deltaTime, 0);
21.                   transform.localScale -= Vector3.one * Time.deltaTime;
22.
23.                   //إن لم يسبق لنا استدعاء Destroy() لنقم باستدعائها الآن
24.                   if(!destroyed){
25.                         //قم بتأجيل التدمير ثانية واحدة حتى يتسنى
26.                         //للاعب مشاهدة الحركة
27.                         Destroy(gameObject, 1);
28.                         //قم بتعديل قيمة المتغير destroyed
29.                         //حتى تمنع استدعاء Destroy() أكثر من مرة
30.                         destroyed = true;
31.                   }
32.             }
33.       }
34. }
```

السرد 15: البريمج الخاص بالهدف

لاحظ أن كل ما يقوم به البريمج هو فحص القيمة hit ومن ثم تدمير الكائن في حال كانت قيمة المتغير true، لاحظ أيضا أنه لا يوجد أي سطر في البريمج يقوم بتعديل قيمة المتغير المذكور. معنى ذلك أن الهدف سيبقى على حاله طالما لم يقم بريمج آخر بتغيير قيمة hit. سنرى بعد قليل كيف ستقوم الطلقة بفحص التصادم بينها وبين الهدف وتعديل القيمة في حال وُجد هذا التصادم. يقوم البريمج أيضا بإضافة نوع من المؤثرات للكائن وهو تدويره وتصغير حجمه بمرور الوقت حالما تتم إصابته، وهذا يجعل الهدف يبدو وكأنه يسقط نحو الأسفل.

التدوير يتم باستخدام الدّالّة transform.Rotate() والتي سبق لنا التعامل معها عدة مرات، أما تصغير الحجم فيتم بطرح قيمة بسيطة من القياس في كل إطار وتساوي متجها تساوي قيم مكوّناته Time.deltaTime، أي أن عملية التصغير تتم بسرعة تعادل مترا واحدا في الثانية مما يجعل الهدف يختفي بعد ثانية واحد حيث سيصبح حجمه صفرا. لهذا السبب نقوم في السطر 27 باستدعاء الدّالّة Destroy() ونعطيها الكائن الذي نريد تدميره (وهو كائن الهدف نفسه في هذه الحالة ونصل إليه عن طريق المتغير gameObject). بالإضافة للكائن الذي سنقوم بتدميره يمكننا أن نزود Destroy() بالوقت الذي عليها انتظاره قبل تنفيذ التدمير وهو هنا ثانية واحدة.

المقصود بتدمير الكائن هنا هو حذفه من المشهد تماما، ويمكن ملاحظة ذلك باختفاء الكائن المدمر من هرمية المشهد. لضمان عدم استدعاء Destroy() أكثر من مرة، استخدمنا المتغير destroyed والذي

نستخدمه كعلامة لقيامنا بعملية الاستدعاء مسبقا. لاحظ أننا هنا احتجنا للمتغير destroyed بسبب تأجيل التدمير لإظهار الحركة، مما سيؤدي لاستدعاء ()Update عدة مرات خلال الثانية القادمة قبل أن يتم التدمير نهائيا. خلال هذه الثانية ما نريده هو تنفيذ حركة السقوط فقط، أما استدعاء ()Destroyed فنريده أن يحدث مرة واحدة فقط.

لنقم الآن بإضافة بريمج آخر على كائن الهدف وستكون مهمته تحريك الهدف حتى لا يظل ساكنا في مكانه. هذه الحركة ستعتمد على الوقت فقط وليس على مدخلات اللاعب كما فعلنا عدة مرات، وذلك لأن اللاعب لا يتحكم بالأهداف بل تتحرك من تلقاء نفسها. سيحرك البريمج الهدف في اتجاه محدد وبسرعة ثابتة. السرد 16 يوضح البريمج AutoMover والذي يقوم بتحريك الكائن تلقائيا حسب السرعة المحدد في متجه السرعة. حين يغادر الكائن مجال رؤية الكاميرا من جهة ما سنقوم بإعادته الجهة المقابلة. هذه الحركة تسمى الالتفاف wrapping وهي مشهورة في كثير من الألعاب كما في اللعبة الكلاسيكية الشهيرة Pac-Man. السرد 17 يوضح البريمج Wrapper والذي يقوم بتدوير الكائن حول الشاشة بناء على موقعه على المحورين x و z.

```
using UnityEngine;
using System.Collections;

public class AutoMover : MonoBehaviour {

	//سرعة الحركة على المحاور الثلاث
	public Vector3 speed = new Vector3(0, 0, 0);

	void Start () {

	}

	void Update () {
		//نقوم بتحريك الكائن بناء على السرعة المحددة
		transform.Translate(speed * Time.deltaTime);
	}
}
```

السرد 16: بريمج يقوم بتحريك الكائن بسرعة ثابتة وبشكل تلقائي

```
using UnityEngine;
using System.Collections;

public class Wrapper : MonoBehaviour {

	//عندما يتجاوز الكائن هذه الحدود على المحاور المختلفة
	//سنقوم بتدويره حول المشهد وإدخاله من الجهة المقابلة
	public Vector3 limits = new Vector3(10, 0, 10);

	void Start () {

	}

```

```
	void Update () {
		//نقوم بأخذ الموقع الحالي
		Vector3 newPos = transform.position;

		if(transform.position.x > limits.x){
			//الكائن تجاوز الحد الأيمن بالتالي نعيده لليسار
			newPos.x = -limits.x;
		}

		if(transform.position.x < -limits.x){
			//الكائن تجاوز الحد الأيسر بالتالي نعيده لليمين
			newPos.x = limits.x;
		}

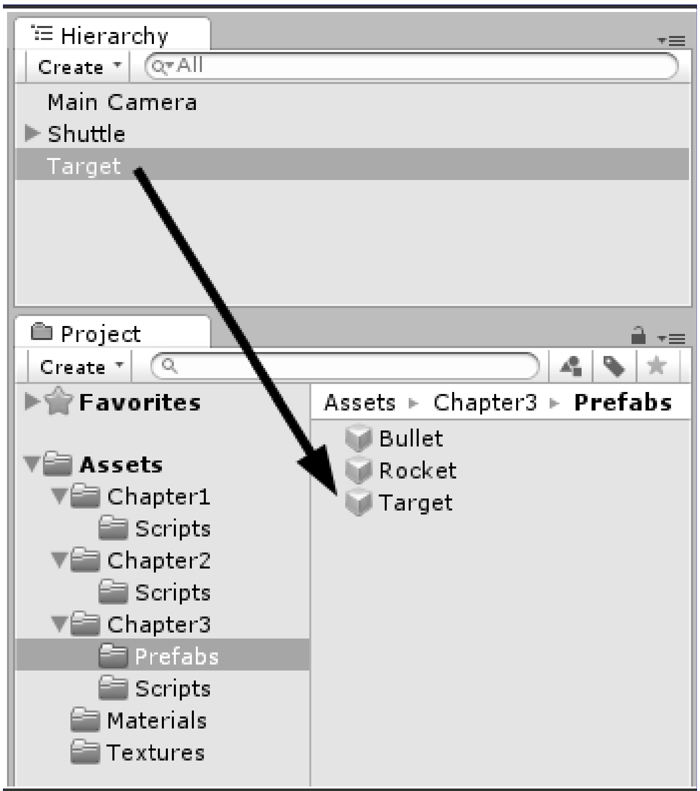
		if(transform.position.z > limits.z){
			//الكائن تجاوز الحد نحو الأمام فنعيده للخلف
			newPos.z = -limits.z;
		}

		//نقوم بتحديث موقع الكائن حسب القيم الجديدة
		transform.position = newPos;
	}
}
```

السرد 17: البريمج الخاص بالالتفاف حول المشهد عند خروج الكائن خارج حدود الكاميرا

لا جديد كما تلاحظ في هذين البريمجين سوى عملية الالتفاف، وهي ببساطة تغيير قيمة x أو z في موقع الكائن بناء على تجاوزها حدودا معينة. بعد إضافة البريمجات Target و AutoMover و Wrapper إلى المكعب الذي سنستعمله كهدف، أصبحنا جاهزين الآن لبناء القالب الخاص بالأهداف، مما سيجعلنا قادرين على إضافة عدة أهداف إلى المشهد.

> لبناء قالب اختر أولا الكائن المرغوب من داخل هرمية المشهد، ومن ثم قم بسحبه من هرمية المشهد إلى أي مجلد داخل مستعرض المشروع، ويفضل أن يكون هناك مجلد خاص بالقوالب يحمل الاسم prefabs كما في الشكل 31. بعدها يمكنك إضافة ما ترغب من نسخ للكائن عن طريق سحب القالب إلى نافذة المشهد أو هرمية المشهد. الكائنات المرتبطة بالقالب ستظهر في الهرمية بلون أزرق، وأي تعديل على القالب كإضافة بريمج أو التعديل على أي من المكوّنات سينعكس على جميع الكائنات المرتبطة بهذا القالب.



الشكل 31: كيفية استخدام كائن لإنشاء قالب جديد

سأتطرق الآن لموضوع مهم يتعلق بكيفية تصميم البرمجيات أثناء التعامل مع Unity. في العادة نستخدم البرمجة غرضية التوجه Object-Oriented Programming مع الوراثة Inheritance وذلك لجعل عملية إعادة الاستخدام أكثر سهولة. أما في Unity فلا نستعمل عادة هذا التوجه ونستبدله بالتركيب composition وهو عبارة عن فصل الوظائف والصفات في بريمجات مختلفة وتركيبها مع بعضها البعض بتوافق يحقق الغرض المطلوب من كل كائن. على سبيل المثال لدينا ثلاث وظائف منفصلة وهي: إصابة الهدف، والحركة التلقائية، والالتفاف. بفصل هذه الوظائف عن بعضها، يمكننا مستقبلا بسهولة بناء كائنات قابلة للإصابة لكنها لا تتحرك، وذلك عن طريق استخدام Target فقط، كما يمكننا بناء كائنات تتحرك لكنها غير قابلة للإصابة إذا استخدمت AutoMover فقط، وغيرها من التراكيب المختلفة.

لنعد الآن إلى مشهدنا لبناء كائن الطلقة ومن ثم استعماله لإنشاء قالب. الطلقة التي سنستعملها هي عبارة عن كائن من نوع كرة بقياس (0.25, 0.25, 0.25) وكل ما سنفعله هو أننا سنضيف لهذه الكرة بريمجين: أحدهما Projectile الموضح في السرد 18 ومهمته دفع الكائن للحركة نحو الأمام بمدى محدد، والثاني هو TargetHitDestroyer ومهمته فحص التصادم بين الطلقة وجميع الأهداف الموجودة في المشهد حتى يقوم بتدمير الهدف في حال حدث التصادم. هذا الأخير موضح في السرد 19.

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {

	//سرعة المقذوف مقدرة بمتر \ ثانية
	public float speed = 15;

	//المدى: وهو عدد الأمتار التي يمكن للمقذوف قطعها قبل أن يتم تدميره
	public float range = 20;

	//نستخدم هذا المتغير لحساب المسافة الكلية التي قطعها المقذوف منذ إطلاقه
	//عندما تتجاوز هذه القيمة المدى المحدد للمقذوف نقوم بتدميره
	float totalDistance = 0;

	void Start () {

	}

	void Update () {
		//المسافة التي سيقطعها المقذوف خلال الإطار الحالي
		float distance = speed * Time.deltaTime;
		//نقوم بتحريك المقذوف نحو الأمام على محوره المحلي z
		transform.Translate(0, 0, distance);

		//نقوم بإضافة المسافة المقطوعة خلال الإطار الحالي إلى المسافة الكلية
		totalDistance += distance;

		//عندما تتجاوز المسافة الكلية المدى المحدد للمقذوف
		//يجب أن نقوم بتدمير المقذوف
		if(totalDistance > range){
			Destroy(gameObject);
		}
	}
}
```

السرد 18: البريمج الخاص بتحريك المقذوف

```
using UnityEngine;
using System.Collections;

public class TargetHitDestroyer : MonoBehaviour {

	void Start () {

	}

	void Update () {
		//البحث عن جميع الكائنات التي تحتوي على البريمج Target
		Target[] allTargets = FindObjectsOfType<Target>();

		//فحص التصادم بين المقذوف وجميع الأهداف
		foreach(Target t in allTargets){
			//نهتم فقط بأمر الأهداف التي لم يسبق وتمت إصابتها
```

```
17.                 if(!t.hit){
18.                     //نقيس المسافة بين الهدف والمقذوف
19.                     float distance = Vector3.Distance(
20.                                     transform.position,
21.                                     t.transform.position);
22.
23.                     if(distance <
24.                         t.transform.localScale.magnitude * 0.5f){
25.                         //hit المقذوف يتصادم مع الهدف، لذا سنقوم بتغيير قيمة
26.                         //true في المقذوف إلى
27.                         t.hit = true;
28.
29.                         //أخيرا نقوم بتدمير المقذوف
30.                         Destroy(gameObject);
31.
32.                     }
33.                 }
34.             }
35.         }
36. }
```

السرد 19: البريمج الخاص بكشف التصادم بين المقذوف والهدف

لنقم الآن بالتعرف على بعض التقنيات الجديدة بالنسبة لنا والتي نراها في السرد 19. في السطر 12 قمنا بتعرف المصفوفة allTargets من نوع Target. المصفوفة هنا هي عبارة عن مجموعة كائنات من نفس النوع مرتبة بشكل متتابع على شكل سلسلة، ويمكن أن نصل لها عن طريق متغير واحد. القوسان [] يدلان دائما على أن المتغير الذي تراه هو مصفوفة أي مجموعة من الكائنات وليس كائنا واحدا. معنى أن هذه المصفوفة من نوع Target أي أن جميع العناصر الموجودة بداخلها هي من هذا النوع. أي أن لدينا الآن مكانا لتخزين مجموعة من الأهداف.

في السطر 12 أيضا قمنا باستدعاء الدّالّة <FindObjectsOfType<Target والتي تقوم بالبحث في المشهد عن جميع الكائنات التي تحتوي على البريمج Target وتعيدها لنا على شكل مصفوفة لنقوم بتخزينها في allTargets. الخطوة التالية في السطر 15 هي استخدام الحلقة التكرارية foreach والتي تسهل علينا التعامل مع المصفوفات وغيرها من مجموعات الكائنات. في الحلقة التكرارية قمنا بتعريف المتغير t والذي ستتغير قيمته في كل دورة حتى تمر على جميع عناصر المصفوفة. بمعنى أن الخطوات من السطر 17 إلى السطر 32 ستتكرر على جميع الأهداف في المشهد. فإذا كان لدينا عشرة أهداف ستتكرر هذه الخطوات عشر مرات كحد أقصى وإن كان لدينا عشرون فعشرون وهلم جرا.

الخطوة الأولى التي سنطبقها على كل الأهداف كما في السطر 17 هي أن نفحص قيمة المتغير hit في هذا الهدف، فإذا كانت قيمته true دل ذلك على أن الهدف قد تمت إصابته مسبقا وبدأ بتشغيل حركة السقوط بالتالي لا يلزمنا عمل أي شيء بخصوصه، لهذا السبب بنينا كافة الخطوات اللاحقة على كون قيمة hit هي false. جدير بالذكر أن <FindObjectsOfType<Target تعطينا جميع الأهداف في المشهد حتى لو كان الهدف مصابا، الاستثناء الوحيد هي الأهداف التي تمت إصابتها قبل أكثر من ثانية

وانتهى تشغيل حركة السقوط وتم حذفها من المشهد تماما.

في الأسطر من 19 إلى 21 قمنا بحساب المسافة بين الهدف والكائن الحالي وهو بطبيعة الحال مقذوف. بعد ذلك نقارن المسافة بين المقذوف والهدف بالقيمة t.transform.localScale.magnitude والتي تمثل طول أحد أضلاع المكعب ثم نضرب هذه القيمة في 0.5 وذلك حتى تعطينا قيمة تقريبية للمسافة بين مركز المكعب وسطحه. قلت "تقريبية" لأن المكعب ليس له مسافة ثابتة بين مركزه وجميع النقاط على سطحه، فالنقاط القريبة من منتصفات وجوه المكعب تكون أقرب إلى المركز من تلك التي على أطراف الأوجه. على أي حال ستعطينا هذه الطريقة نتيجة مقبولة للعبتنا وهذا لا ينفي أننا سنتعرف على طرق أخرى أكثر دقة. إذا تحقق الشرط يعني ذلك أن المسافة بين المقذوف والهدف قصيرة كفاية لتحتسب على أنها تصادم، بالتالي نقوم في السطر 27 بتغيير قيمة المتغير hit إلى true حتى يدخل الهدف في حركة السقوط، ومن ثم نقوم في السطر 30 بتدمير كائن الطلقة فوريا حتى لا يتعدى هذا الهدف لغيره.

بعد أن تضيف هذين البريمجين إلى الكرة، يمكنك أن تقوم بإنشاء القالب الخاص بالطلقة، والذي سنقوم لاحقا بإنشاء نسخ منه عند إطلاق النار من المركبة. بما أننا لا نحتاج لوجود طلقة في المشهد عند بداية اللعب، حيث أن إنشاءها يعتمد على مدخلات اللاعب، علينا أن نقوم بحذف الطلقة من المشهد بعد إنشاء القالب.

> لحذف كائن من المشهد، قم ببساطة باختياره من هرمية المشهد ومن ثم اضغط Delete على لوحة المفاتيح.

الخطوة التالية هي إنشاء قالب الصاروخ، وهو مثل الطلقة لكن له القدرة على ملاحقة الهدف بخلاف الطلقة التي تسير بخط مستقيم فقط. لتمثيل الصاروخ قم بإنشاء مكعب بحجم (0.1, 0.1, 0.75) ومن ثم أضف له البريمجين Projectile و TargetHitDestroyer وذلك حتى يتحرك للأمام ويدمر الأهداف حين يصيبها. الوظيفة الثالثة التي يقوم بها الصاروخ هي البحث عن أقرب هدف له لحظة الانطلاق والتركيز عليه وملاحقته إلى أن يصيبه أو يقطع المدى المحدد له دون إصابته بالتالي يدمر الصاروخ تلقائيا. هذه الوظيفة سنكتبها في البريمج TargetFollower والذي يظهره السرد 20.

```
using UnityEngine;
using System.Collections;

public class TargetFollower : MonoBehaviour {

	//الهدف الذي سنقوم بتتبعه
	Target currentTarget;

	void Start () {
		//سنقوم بالبحث في كافة الأهداف عن أقربها إلينا
		Target[] allTargets = FindObjectsOfType<Target>();

```

```
        //تحقق من وجود أهداف في المشهد
        if(allTargets.Length > 0){
            //سنفرض مبدئيا أن الهدف الأول في المصفوفة هو الأقرب إلينا
            Target nearest = allTargets[0];

            //نبدأ الآن بالبحث عن أقرب هدف
            foreach(Target t in allTargets){
                //إن كان الهدف مصابا من قبل
                //لن نهتم لأمره
                if(!t.hit){
                    //المسافة بين المقذوف
                    //t والهدف الحالي
                    float distance =
                        Vector3.Distance(
                            transform.position,
                            t.transform.position);

                    //نحسب المسافة بين المقذوف
                    //nearest والهدف الأقرب
                    float minDistance =
                        Vector3.Distance(
                            transform.position,
                            nearest.transform.position);

                    //نقوم بتحديث قيمة الهدف الأقرب إن لزم الأمر
                    if(distance < minDistance){
                        nearest = t;
                    }
                }
            }

            //نحدد الهدف الأقرب على أنه الهدف الحالي
            currentTarget = nearest;
        }
    }

    void Update () {
        //نتأكد أولا من أن الهدف الحالي الذي نتبعه موجود أصلا ولم تتم إصابته
        //من قبل صاروخ أو طلقة أخرى
        if(currentTarget != null && !currentTarget.hit){
            //نقوم بتدوير المقذوف لينظر إلى الهدف الذي نتبعه
            transform.LookAt(currentTarget.transform.position);
        }
    }
}
```

السرد 20: البريمج الذي يسمح للصاروخ بالبحث عن أقرب هدف وتتبعه

في البداية قمنا في الدّالّة ()Start بالبحث عن أقرب الأهداف إلى المقذوف حتى يتم تعيينه كهدف للتتبع. طريقة البحث خطية ومعروفة لدى من لديه بعض الخبرة في البرمجة. المسألة التي نتعامل معها هي أن لدينا مجموعة من الأهداف مخزنة في المصفوفة allTargets، ونريد أن نعرف أي هذه الأهداف هو الأقرب مسافة إلينا. لكن وقبل ذلك كله ينبغي أن نتأكد من وجود عناصر في المصفوفة

أصلا، أي أنه هناك أهدافا لا تزال موجودة في المشهد. هذه الخطوة ننفذها في السطر 14 حيث يمكننا أن نعرف عدد العناصر الموجودة في المصفوفة عن طريق allTargets.Length ونفحص إن كان عددها أكبر من صفر.

بعد ذلك نقوم مبدئيا بافتراض أن أقرب الأهداف هو هو الأول في المصفوفة والذي نصل إليه عن طريق تحديد الموقع كما في allTargets[0]. وهذا يعني أنني أريد أن أستخرج العنصر الأول في المصفوفة. في معظم لغات البرمجة ومن ضمنها #C التي نستعملها، يكون الموقع الأول في المصفوفة هو صفر، فإذا كان في المصفوفة 10 عناصر مثلا، سيكون العنصر الأخير في الموقع 9. نقوم بتخزين الهدف الأول في المتغير nearest وتعني بالعربية الأقرب، بعدها ندخل في حلقة تكرارية تمر على جميع الأهداف الموجودة في المصفوفة وذلك في السطر 19. وبما أننا نهتم فقط بالأهداف التي لم تصب حتى الآن، نقوم في السطر 22 بالتأكد من عدم إصابة الهدف قبل أن نقوم بحساب أي مسافة بينه وبين المقذوف.

المتغير distance الذي نعرفه في السطر 25 نخزن فيه المسافة بين الهدف الحالي وبين المقذوف، بينما نخزن في المتغير minDistance في السطر 32 المسافة بين المقذوف وبين الهدف الذي نفترض أنه الأقرب حتى الآن. بعد ذلك نقارن هاتين المسافتين في السطر 38، فإن تبين لنا أن المسافة distance أقل من minDistance فهذا يعني أن الهدف الحالي أقرب مسافة من الهدف المخزن حاليا في nearest، بالتالي نقوم بتحديث قيمة nearest لتأخذ القيمة الجديدة، وهكذا نمر على كل الأهداف في المصفوفة، لنحصل في النهاية على الهدف الأقرب مخزنا في المتغير nearest. بعد الانتهاء من قياس المسافات ومقارنتها نقوم أخيرا في السطر 45 بتخزين قيمة nearest في currentTarget حتى يصبح هو الهدف الذي نتتبعه.

الدّالّة ()Update تقوم بتنفيذ عملية التتبع على خطوتين، الخطوة الأولى هي أن تتأكد من وجود هدف أصلا حتى تقوم بتتبعه، وذلك في السطر 52 حيث تتأكد من أن قيمة currentTarget لا تساوي null. المقصود بـ null هو عدم وجود قيمة، أي أن currentTarget عديم القيمة ولم يتم تخزين أي هدف فيه. هذا الأمر ممكن الحدوث في حال تم إطلاق الصاروخ دون وجود أي هدف في المشهد مما يجعل الهدف الأقرب غير موجود أصلا. الخطوة الثانية هي أن تتأكد من أن الهدف الذي تقوم بتتبعه لا زال موجودا ولم تتم إصابته عن طريق طلقة أخرى أو صاروخ آخر. عندما يتحقق هذان الشرطان يتم توجيه المقذوف نحو الهدف عن طريق ()transform.LookAt. هناك ملاحظة مهمة أود الإشارة إليها هنا وهي أن العملية && تسمح بتنفيذ جملة if فقط في حال تحقق الشرطان على طرفي العملية، وفي حالة لم يتحقق الشرط الأول، فإنها لا تحاول أن تفحص الشرط الثاني. مبدأ العمل هذا مهم بالنسبة لنا في هذه الحالة، حيث لا يمكننا أن نفحص قيمة currentTarget.hit في الوقت الذي تكون فيه currentTarget نفسها عديمة القيمة، وإلا فإن خطأ سيحدث أثناء التشغيل.

بعد إضافة TargetFollower إلى الصاروخ أصبح جاهزا لاستخدامه لبناء القالب، ولا تنسى حذف كائن

الصاروخ المستخدم من المشهد بعد عمل القالب. بالتالي أصبح لدينا الآن ثلاث قوالب وهي الهدف Target والصاروخ Rocket والطلقة Bullet. لإكمال بناء المشهد، قم بإضافة مجموعة من الأهداف وأعطها سرعات متفاوتة. يمكن مثلا إضافة صفين من الأهداف وإعطاء عناصر أحدهما السرعة (3, 0, 0) في البريمج AutoMover حتى تتحرك نحو اليمين والصف الآخر السرعة (-3, 0, 0) حتى تتحرك نحو اليسار.

لننتقل الآن إلى بريمج التحكم بالمركبة وهو بريمج بسيط كما ترى في السرد 21. ويقوم بوظيفة واحدة هي قراءة حالة مفاتيح الأسهم وتحريك المركبة على المحورين x و z بناء عليها.

```
using UnityEngine;
using System.Collections;

public class ShuttleControl : MonoBehaviour {

	//سرعة حركة المركبة
	public float speed = 7;

	void Start () {

	}

	void Update () {
		//قراءة حالة مفاتيح الأسهم
		//وتحريك المركبة بناء عليها
		if(Input.GetKey(KeyCode.UpArrow)){
			transform.Translate(0, 0, speed * Time.deltaTime);
		} else if(Input.GetKey(KeyCode.DownArrow)){
			transform.Translate(0, 0, -speed * Time.deltaTime);
		}

		if(Input.GetKey(KeyCode.RightArrow)){
			transform.Translate(speed * Time.deltaTime, 0, 0);
		} else if(Input.GetKey(KeyCode.LeftArrow)){
			transform.Translate(-speed * Time.deltaTime, 0, 0);
		}
	}
}
```

السرد 21: بريمج التحكم بالمركبة

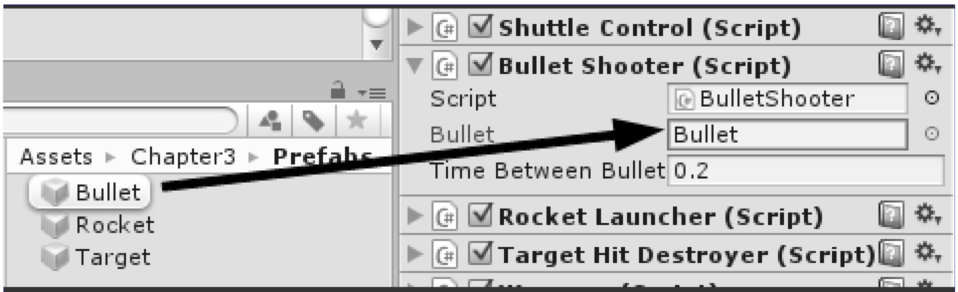
بالإضافة إلى ShuttleControl سأقوم أيضا بإضافة كل من البريمجين Wrapper و TargetHitDestroyer إلى كائن المركبة، مما سيجعلها تلتف من اليمين إلى اليسار وبالعكس، بالإضافة إلى التفافها من الأمام إلى الخلف كما سبق شرحه في السرد 17. إضافة TargetHitDestroyer ستجعل المركبة تتحطم إذا لامست أحد الأهداف (أي أن كائنها سيتم تدميره وحذفه من المشهد). بالإضافة إلى هذه البريمجات الثلاثة، سنضيف كلا من BulletShooter و RocketLauncher. مهمة هذين البريمجين متشابهة، حيث أنهما يقومان بإضافة كائن جديد إلى المشهد (طلقة أو صاروخ) بناء على مدخلات

اللاعب، وهناك اختلافات بينهما من حيث كيفية تكرار عملية الإطلاق كما سنرى. لنبدأ مع BulletShooter والموضح في السرد 22.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class BulletShooter : MonoBehaviour {
5.
6.        //القالب الذي سنستخدمه لبناء الطلقات الجديدة
7.        public GameObject bullet;
8.
9.        //كم من الثواني يجب أن تمر بين كل طلقتين متتابعتين؟
10.       public float timeBetweenBullets = 0.2f;
11.
12.       //متى قامت المركبة بإطلاق آخر طلقة؟
13.       float lastBulletTime = 0;
14.
15.       void Start () {
16.
17.       }
18.
19.       void Update () {
20.              //سنستخدم مفتاح control الأيسر لإطلاق الطلقات
21.              if(Input.GetKey(KeyCode.LeftControl)){
22.                     //هل مضى الوقت الكافي منذ آخر طلقة تم أطلاقها؟
23.                     if(Time.time - lastBulletTime > timeBetweenBullets){
24.                            //نقوم بإنشاء طلقة جديدة مستخدمين القالب
25.                            //موقع الطلقة هو نفس موقع المركبة الحالي
26.                            //وستنظر الطلقة في نفس الاتجاه الذي تنظر إليه المركبة
27.                            Instantiate(bullet, //الكائن الذي سنقوم بإنشائه
28.                               transform.position, //موقع الكائن
29.                               transform.rotation);//دوران الكائن
30.
31.                            //نقوم بتسجيل الوقت الذي تمت فيه عملية إطلاق الطلقة
32.                            lastBulletTime = Time.time;
33.                     }
34.              }
35.       }
36. }
```

السرد 22: بريمج إطلاق الطلقات من المركبة

بعد إضافة هذا البريمج إلى كائن المركبة، أول ما يتوجب علينا فعله هو أن نخبر البريمج عن القالب الذي ينبغي عليه استخدامه حين يقوم بإنشاء طلقة جديدة. تذكر أننا سبق وبنينا كائن الطلقة وأضفنا له البريمجات اللازمة وأنشأنا منه قالبا. سنقوم الآن بربط هذا القالب بالمتغير bullet الذي عرّفناه في السطر 7 من البريمج BulletShooter، وذلك عن طريق سحبه من مستعرض المشروع إلى خانة المتغير في شاشة الخصائص كما في الشكل 32.



الشكل 32: كيفية ربط القالب بمتغير داخل البريمج

إضافة إلى إنشاء نسخ من قالب الطلقة، يقوم هذا البريمج بحساب الوقت المنقضي منذ آخر طلقة تم إطلاقها والتأكد من وجود تردد أعلى للطلقات لا يمكن تجاوزه. هذا التردد نقوم بتحديده عن طريق المتغير timeBetweenBullets في السطر 10 والذي أعطيناه افتراضيا القيمة 0.2، أي أننا سنسمح بطلقة واحدة كل 0.2 ثانية أي خمس طلقات في الثانية كحد أقصى. عملية الإطلاق تتم عن طريق ضغط اللاعب على مفتاح control الأيسر كما هو واضح في السطر 21، بعدها يقوم البريمج بطرح وقت آخر طلقة من الوقت الحالي للتأكد من مرور وقت كاف بين الطلقتين كما في السطر 23.

بعد هذا قمنا باستدعاء دالّة جديدة هي ()Instantiate، والتي تقوم بعمل نسخة جديدة من القالب في كل مرة نستدعيها. إضافة إلى القالب bullet، قمنا بتزويد الدّالّة بكل من موقع إنشاء النسخة الجديدة ودورانها، وهي في هذه الحالة نفس موقع المركبة ونفس دورانها، مما يجعل الطلقة تبدو وكأنها خرجت من المركبة ويجعلها تنظر تلقائيا في اتجاه مقدمة المركبة. بعد بناء النسخة الجديدة أصبح لدينا كائن طلقة جديد في المشهد، وهذا الكائن يحتوي على كافّة البريمجات التي أضفناها للقالب Bullet، بالتالي سيقوم هذا الكائن بتنفيذ السلوك المطلوب حيث سيتحرك نحو الأمام ويحسب المدى المقطوع ويفحص التصادم مع الأهداف. أخيرا نقوم في السطر 32 بتسجيل الوقت الذي تم فيه إطلاق الطلقة حتى نتمكن من حساب تردد الطلقات ومنع تكرار الإطلاق بوتيرة أعلى من الحد الأقصى.

البريمج الثاني الذي سنضيفه هو RocketLanuncher والخاص بإطلاق الصواريخ. هذا البريمج شبيه بسابقه باستثناء بعض الفروقات. السرد 23 يوضح هذا البريمج.

```
using UnityEngine;
using System.Collections;

public class RocketLauncher : MonoBehaviour {

        //القالب الذي سنستخدمه لإنشاء الصواريخ
        public GameObject rocket;

        void Start () {

        }
```

```

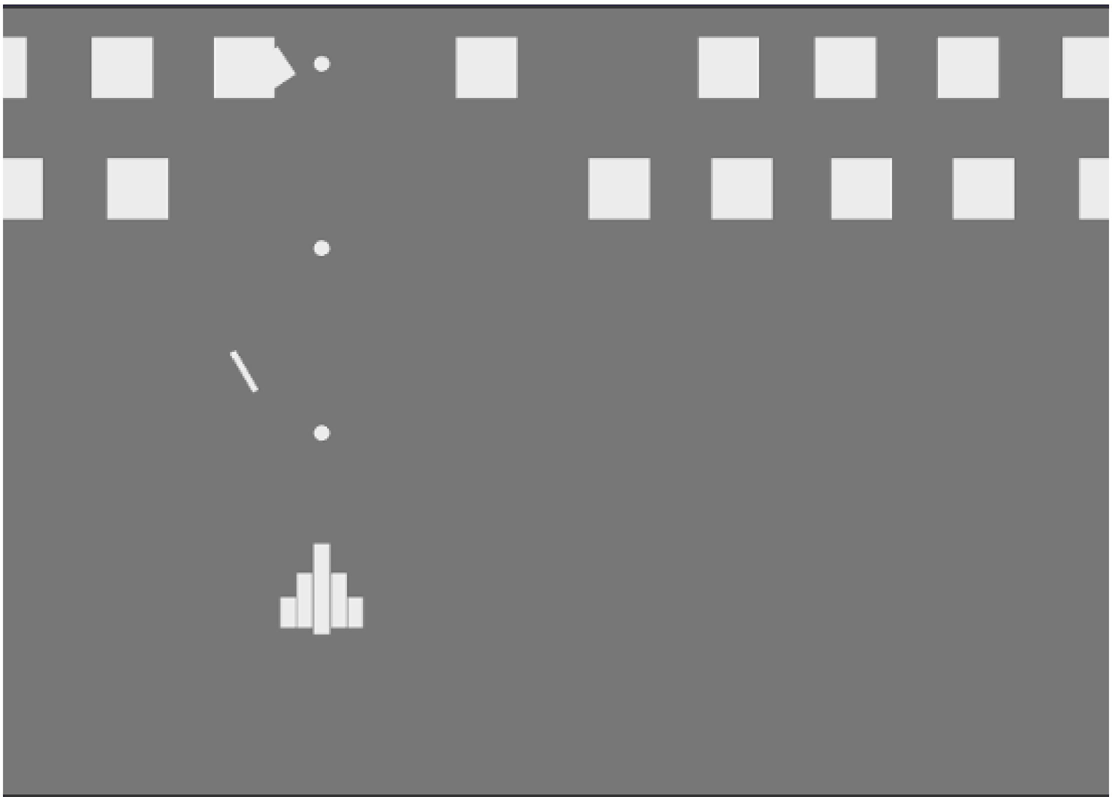
	void Update () {
		//نستخدم مفتاح المسافة لإطلاق الصواريخ دون السماح بالضغط المستمر عليه
		if(Input.GetKeyDown(KeyCode.Space)){

			//كم عدد الصواريخ الموجودة في المشهد؟
			TargetFollower[] rockets =
				FindObjectsOfType<TargetFollower>();

			//لا نسمح بوجود أكثر من صاروخ في المشهد في نفس الوقت
			if(rockets.Length == 0){
				//قم بإنشاء صاروخ جديد في نفس موقع المركبة
				//باستخدام نفس دورانها
				Instantiate(rocket,
					transform.position, transform.rotation);
			}
		}
	}
}
```

السرد 23: بريمج إطلاق الصواريخ من المركبة

الملاحظة المهمة في هذا البريمج هي أنه لا يسمح بإطلاق الصاروخ طالما وُجد صاروخ آخر في المشهد. يتم التعرف على عدد الصواريخ الموجودة في المشهد عن طريق البحث عن كائنات تحتوي على البريمج TargetFollower، السبب في اختيار هذا البريمج تحديدا هو أنه لا يوجد إلّا في كائن الصاروخ، بالتالي فإن عدد الكائنات في المشهد التي تحتوي على هذا البريمج هو نفس عدد الصواريخ بالتأكيد. أما لو قمنا بالبحث عن كائنات من نوع Projectile مثلا، فإننا سنحصل على كائنات الطلقات والصواريخ معا وهو ما لا نريده هنا. الشكل 33 يعرض صورة من اللعبة بعد إتمامها، ويمكنك مشاهدة النتيجة أيضا في المشهد scene9 في المشروع المرفق.



الشكل 33: لعبة المركبة الخاصة بعرض كيفية برمجة المقذوفات والطلقات

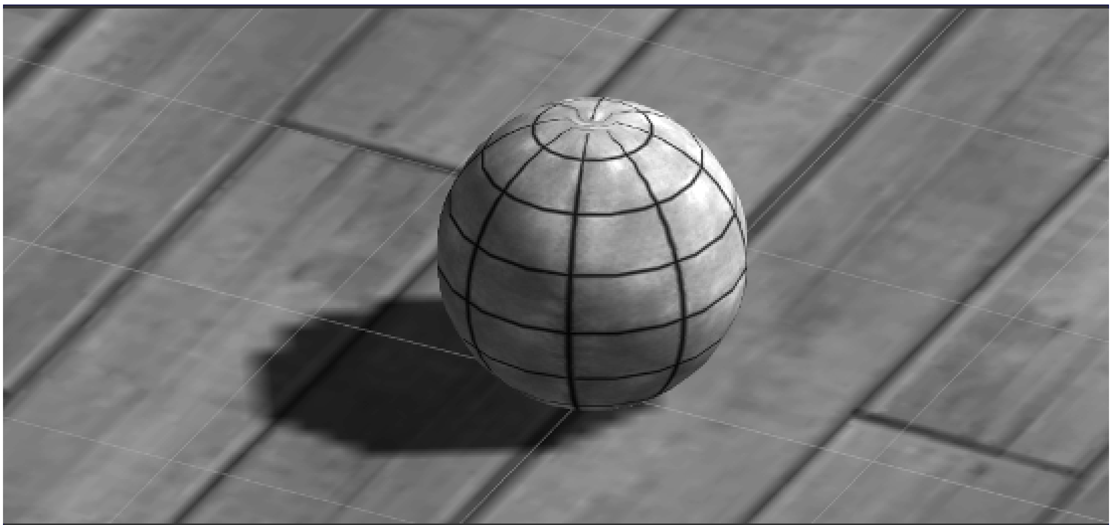
الفصل الثاني: برمجة الأشياء القابلة للجمع

تعتمد الكثير من ألعاب الفيديو على نشر مجموعة من القطع النقدية أو غيرها من الأشياء ذات القيمة داخل اللعبة، وذلك من باب تشجيع اللاعب على استكشاف عالم اللعبة بشكل أكبر. هذه الأشياء يمكن أن تستخدم لاحقا في تحسين أداء اللاعب مثل شراء المعدات أو تعلم مهارات جديدة، خاصة في ألعاب تقمص الأدوار RPG. سنتعلم في هذا الفصل كيف نبرمج آلية تجميع هذه القطع وغيرها.

عند الحديث عن الأشياء القابلة للجمع، فلا بد من الإشارة إلى سلوك مشترك بين جميع هذه الأشياء بغض النظر عن ماهيتها أو ما تقدمه للاعب. هذا السلوك هو أنه ببساطة يمكن أن يتم "جمعها" من قبل "المجمّعين" بمجرد لمسها. ذكرت هنا كلمة "مجمّعين" لأن اللاعب ليس بالضرورة الشخص الوحيد القادر على جمع هذه الأشياء في عالم اللعبة، بل يمكن أن تمتلك الشخصيات الأخرى التي يتحكم بها الحاسب هذه القدرة.

بمجرد أن يلمس أحد المجمعين أحد الكائنات القابلة للجمع، فإنه يحاول أن "يأخذ" هذا الكائن أو "يجمعه"، هذه المحاولة قد تنجح وقد تفشل اعتمادا على عدة عوامل كما سنرى. لشرح فكرة المجمّع وما يمكن أن يجمعه، سأستخدم كرة لتعبر عن اللاعب. هذه الكرة ستتدحرج فوق أرضية خشبية ويمكن

للاعب أن يراها من منظور علوي ويحركها باستخدام الأسهم على المحورين x و z. تمتلك هذه الكرة القدرة على جمع القطع النقدية بمجرد لمسها، وتقوم بإضافة هذه القطع إلى حقيبة محتوياتها.
بالإضافة للقطع النقدية، يمكن للكرة أن تجمع نوعين من "الطعام". هذا الطعام يعمل على زيادة حجم الكرة لفترة زمنية محددة، مما يجعل عملية تجميع القطع أسهل وأسرع. لنبدأ إذن بكرة ذات قياس أولي (1, 1, 1) وإكساء زجاجي، وأرضية عبارة عن لوح بقياس (10, 1, 10) ذات إكساء خشبي. بعد بناء المشهد كما في الشكل 34، قم بوضع الكاميرا فوق الكرة تماما وبارتفاع 15 ثم قم بتدويرها لتنظر نحو الأسفل.



الشكل 34: الكرة الزجاجية التي سنستعملها كشخصية للاعب

لنتحكم بالكرة سنقوم بكتابة البريمج BallRoller والموضح في السرد 24. إضافة لذلك سنحتاج للبريمج ObjectTracker والذي سنقوم بإضافته للكاميرا وذلك حتى تتمكن من تتبع حركة الكرة واللحاق بها. السرد 25 يوضح البريمج ObjectTracker. أخيرا، لا تنسى أن تقوم بإسناد كائن الكرة للمتغير objectToTrack في ObjectTracker، وذلك حتى تعرف الكاميرا ما هو الكائن الذي عليها أن تتبعه.

```
using UnityEngine;
using System.Collections;

public class BallRoller : MonoBehaviour {

	//سرعة الحركة على المحورين x و z
	public float moveSpeed = 5;

	//سرعة الدحرجة
	public float rollSpeed = 360;

	void Start () {

	}

```

```
    void Update () {
        //z تحرك على محور الفضاء x وتدرج على محور الفضاء
        if(Input.GetKey(KeyCode.UpArrow)){
            transform.Translate(0, 0,
                moveSpeed * Time.deltaTime, Space.World);

            transform.Rotate(rollSpeed * Time.deltaTime,
                                0, 0, Space.World);
        } else if(Input.GetKey(KeyCode.DownArrow)){
            transform.Translate(0, 0,
                -moveSpeed * Time.deltaTime, Space.World);

            transform.Rotate(-rollSpeed * Time.deltaTime,
                                0, 0, Space.World);
        }

        //x تحرك على محور الفضاء z وتدرج على محور الفضاء
        if(Input.GetKey(KeyCode.RightArrow)){
            transform.Translate(moveSpeed * Time.deltaTime,
                                    0, 0, Space.World);

            transform.Rotate(0, 0,
                -rollSpeed * Time.deltaTime, Space.World);

        } else if(Input.GetKey(KeyCode.LeftArrow)){
            transform.Translate(-moveSpeed * Time.deltaTime,
                                    0, 0, Space.World);

            transform.Rotate(0, 0,
                rollSpeed * Time.deltaTime, Space.World);
        }
    }
}
```

السرد 24: بريمج لتحريك الكرة ودحرجتها باستخدام مفاتيح الأسهم

```
using UnityEngine;
using System.Collections;

public class ObjectTracker : MonoBehaviour {

    //الهدف المراد تتبعه//
    public Transform objectToTrack;

    void Start () {

    }

    void Update () {
        //غير الموقع (x, z) لهذا الكائن إلى الموقع (x, z) للكائن الذي//
        //نقوم بتتبعه//
        Vector3 newPos = transform.position;
        newPos.x = objectToTrack.position.x;
        newPos.z = objectToTrack.position.z;
        transform.position = newPos;
    }
```

```
}
```

السرد 25: البريمج الذي يسمح للكاميرا بتتبع اللاعب من الأعلى

هذه الكرة ستكون هي من يقوم بتجميع الأشياء عن طريق ملامستها. أما ما ستقوم بجمعه فهي قطع نقدية وأشياء أخرى كما سبق وذكرنا. لذا يلزمنا الآن بريمجان آخران هما Collectable والذي سنقوم بإضافته لكل ما يمكن جمعه، و Collector والذي يحتاجه كل كائن سيقوم بعملية الجمع. لنبدأ مع البريمج الأول Collectable والموضح في السرد 26.

```
using UnityEngine;
using System.Collections;

public class Collectable : MonoBehaviour {

	//المسافة بين مركز الكائن
	//وسطح التصادم الخارجي
	public float radius = 0.5f;

	void Start () {

	}

	void Update () {

	}

	//سيقوم المجمّع صاحب البريمج Collector باستدعاء هذه الدالّة
	//عندما يلامس هذا الكائن
	public void Collect(Collector owner){
		//قم بتبليغ البريمجات الأخرى المضافة على هذا الكائن
		//بحدوث التلامس مع المجمّع وذلك حتى تنفذ منطق عملية الجمع
		SendMessage("Collected",
			owner, SendMessageOptions.RequireReceiver);
	}
}
```

السرد 26: البريمج الخاص بالأشياء القابلة للجمع

كما تلاحظ فإن الدالتين ()Start و ()Update خاليتان من أي أوامر، مما يعني أن هذا الكائن سيكون خاملا ولن يقوم بأي شيء دون تدخل من كائن آخر. فكل ما يفعله Collectable هو الانتظار حتى يقوم أحد ما باستدعاء الدّالة ()Collect. فائدة المتغير radius هو أن المجمّع Collector سيقوم باستخدامه لفحص التصادم مع الكائن القابل للجمع. عندما يلمس اللاعب أو المجمّع بشكل عام الكائن المحتوي على البريمج Collectable فإنه يقوم باستدعاء الدالّة ()Collect ويقوم بتزويد نفسه عبر المتغير owner وذلك حتى يعرف البريمج من هو الذي يحاول أن يجمع هذا الكائن. كل ما تقوم به ()Collect هو أنها ترسل الرسالة Collected مرفقة معها owner. إرفاق owner مع الرسالة مهم لمعرفة هوية المجمّع؛ فمثلا إذا كان ما نقوم بجمعه هو قطعة نقدية، فإننا نعرف عن طريق owner من هو الذي قام بجمعها

وبالتالي من المجمّع الذي ينبغي أن نزيد رصيده المالي.

السؤال الذي ينبغي أن نفكر بإجابته الآن هو: عندما يتم إرسال الرسالة Collected، من الذي سيقوم باستقبالها؟ الإجابة ببساطة هي: كافة البريمجات الأخرى المضافة على الكائن القابل للجمع والمحتوي على البريمج Collectable. سنرى بعد قليل كيف يمكن أن نستقبل هذه الرسالة ونربط ذلك بمنطق مخصص لعملية الجمع. معنى ذلك أن البريمج Collectable لا يقوم عمليا بأي شيء سوى إخبار البريمجات الأخرى أن owner يحاول أن يقوم بجمع هذا الكائن. من المهم أن أشير هنا إلى استخدام الخيار SendMessageOptions.RequireReceiver عند إرسال الرسالة. هذا الخيار يؤكد على أنه ينبغي استقبال الرسالة من بريمج واحد على الأقل، ويقوم بالإبلاغ عن خطأ في البرنامج إذا لم يتم هذا الأمر. الهدف من استخدام هذا الخيار هو التأكيد على كون عملية الجمع ذات معنى منطقي، وعدم استقبال هذه الرسالة يجعلها بخلاف ذلك.

بالانتقال إلى الطرف الآخر لعملية الجمع وهو المجمّع، نجد البريمج Collector الذي يقوم بفحص التصادمات وتنفيذ محاولات الجمع. السرد 27 يوضح هذا البريمج والذي سنقوم بإضافته للكرة التي تمثل اللاعب.

```
using UnityEngine;
using System.Collections;

public class Collector : MonoBehaviour {

	//نصف القطر المستخدم لفحص التصادمات
	public float radius = 0.5f;

	void Start () {

	}

	void Update () {
		//ابحث عن كافة الأشياء القابلة للجمع في المشهد
		Collectable[] allCollectables =
					FindObjectsOfType<Collectable>();

		//افحص عملية التصادم مع الأشياء القابلة للجمع
		//يتم هذا الفحص بمقارنة المسافة بين المراكز مع مجموع أنصاف الأقطار
		foreach(Collectable col in allCollectables){
			float distance =
				Vector3.Distance(transform.position,
						col.transform.position);

			//كون المسافة بين المركزين أقل من مجموع أنصاف الأقطار يعني وجود تصادم
			//لذا نحاول أن نقوم بجمع هذا الكائن
			if(distance < col.radius + radius){
				//أخبر الكائن القابل للجمع بأن هذا المجمّع
				//يحاول أن يجمعه
```

```
30.                         col.Collect (this);
31.                   }
32.             }
33.       }
34. }
```

السرد 27: البريمج الخاص بتجميع الكائنات القابلة للجمع

بقراءة هذا البريمج يمكننا الاستنتاج بأنه ذو وظيفة مجرّدة، ولا تُعنى بأي تفاصيل خاصة بعملية الجمع نفسها وكيف تتم. فكل ما يقوم به البريمج هو اكتشاف التصادمات مع الأشياء القابلة للجمع استدعاء الدّالة ()Collect من هذه الأشياء. اكتشاف التصادمات يتم عن طريق حساب المسافة بين المجمّع والكائن المراد جمعه، ومن ثم مقارنة هذه المسافة مع أنصاف الأقطار المفترضة للكائنين كما في السطر 27. ذكرت كلمة "مفترضة" لأن الكائنات ليست بالضرورة دائرية أو كروية الشكل حتى يكون لها نصف قطر. لكن هذه الطريقة تعطي فحص تصادم بدقة كافية لتحقيق مبتغانا في هذا المثال. إذا تم اكتشاف التصادم فإن المجمّع يقوم باستدعاء الدّالة ()Collect من الكائن أو الكائنات التي حدث التصادم معها. لاحظ أن المجمّع يقوم بتزويد نفسه عبر المتغير owner وذلك باستخدام الكلمة this في السطر 30. هذه الكلمة تستخدم كمتغير خاص يمكّن البريمج من قراءة قيمة نفسه داخليا، وهذا ما يحتاجه Collector ليخبر الدّالة ()Collect أنه هو نفسه owner الذي يحاول أن يقوم بعملية الجمع.

ما قمنا ببنائه عمليا حتى هذه اللحظة هو آلية مختصة بتحديد الأشياء التي يمكن جمعها، والكائنات التي يمكنها القيام بعملية الجمع، إضافة إلى آلية لفحص التصادم بينهما وإبلاغ Collectable بأن Collector يحاول القيام بعملية الجمع. على الرغم من ذلك لم نقم حتى الآن بعمل أي شيء يختص بمنطق عملية الجمع وكيف تتم. من الناحية النظرية، فإن أنواع الأشياء التي يمكن جمعها هو غير محدود ويختلف تنفيذه باختلاف ما نجمعه. فمثلا جمع قطعة نقدية تزيد رصيد اللاعب من المال، بينما جمع نوع من الطعام مثلا قد يزيد صحة اللاعب. في المثال الذي سنتطرق إليه لدينا نوعان من الأشياء التي يمكن جمعها: القطع النقدية والطعام. القطع النقدية تزيد الرصيد المالي في حقيبة اللاعب بينما يعمل الطعام على زيادة نصف قطر الكرة لفترة محددة مما يجعل عملية جمع القطع النقدية أسرع. سيكون لدينا نوعان من الطعام: أخضر وأحمر، ويختلفان في مدة التأثير ومقدار الزيادة. قبل التطرق لهذه التفاصيل لنلقي نظرة على بريمج يعمل على تدوير القطع النقدية حول محور الفضاء العمودي. هذا البريمج هو YRotator وموضح في السرد 28.

```
using UnityEngine;
using System.Collections;

public class YRotator : MonoBehaviour {

      //سرعة التدوير مقدرة بدرجة \ ثانية
      public float speed = 180;

      //هل يجب أن يبدأ التدوير بزاوية عشوائية؟
      public bool randomStartAngle = true;
```

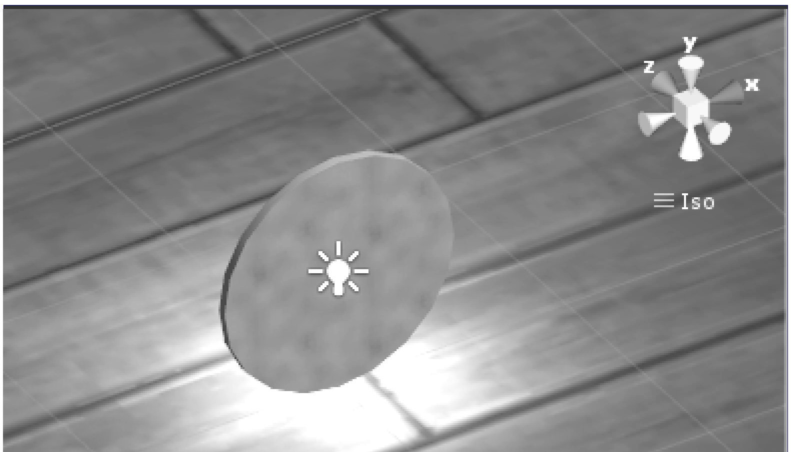
```
	void Start () {
		if(randomStartAngle){
			//قم بالتدوير بزاوية عشوائية بين 0 و 180 درجة
			transform.Rotate(
				0, Random.Range(0, 180), 0, Space.World);
		}
	}

	void Update () {
		//قم بالتدوير حول المحور y بمرور الوقت
		transform.Rotate(
			0, speed * Time.deltaTime, 0, Space.World);
	}
}
```

السرد 28: بريمج لتدوير الكائن حول محور الفضاء y اعتمادا على الوقت

بمقدورنا أن نحدد randomStartAngle لتجنب الحصول على دوران موحد لجميع القطع النقدية، مما يعطي المشهد عشوائية جميلة الشكل. لاحظ أننا استخدمنا لهذا الغرض الدّالة ()Random.Range والتي يمكنها أن تعطينا قيما عشوائية بين الحدين الأدنى والأعلى الذين نقوم بتزويدهما. ففي هذه الحالة تعطينا قيمة عشوائية للزاوية المبدئية بين 0 و 180 درجة.

ما يتوجب علينا القيام به الآن هو إضافة البريمجين Collectable و YRotator إلى كافّة الكائنات التي نرغب بجعلها قابلة للجمع، حيث يمكن القول أننا سنستخدم التدوير كعلامة مميزة تدل اللاعب على ما يمكنه جمعه من عناصر المشهد. لنبدأ مع القطع النقدية والتي هي عبارة عن أسطوانة بقياس (1 ,0.02 ,1) مضافا إليها إكساء بلون الذهب لتعطي المظهر المطلوب. كما يمكننا أن نجعلها تشع بإضافة ضوء نقطي كابن لهذه الأسطوانة وإعطائه لونا أصفر. بعد بناء هذه الأسطوانة كما في الشكل 35، علينا أن نضيف إليها البريمجين Collectable و YRotator ومن ثم نقوم ببناء قالب منها، ذلك لأننا سنضيف عددا لا بأس به من القطع النقدية للمشهد.



الشكل 35: القطعة النقدية التي سنستخدمها في المثال

بإضافة هذين البريمجين إلى القطعة النقدية ستستدير هذه القطعة عند تشغيل اللعبة. إضافة إلى ذلك فإنها ستكون قابلة للجمع بمعنى أن التصادم بينها وبين المجمّع (اللاعب في هذه الحالة) سيتم اكتشافه. بيد أن هذا التصادم ليس له أي أثر حتى هذه اللحظة، كونه يؤدي إلى إرسال الرسالة Collected والتي لا يوجد حتى الآن من يستقبلها. عند استقبال هذه الرسالة من قبل قطعة النقود فإنها تقوم بزيادة الرصيد المالي في حقيبة اللاعب أو المجمّع بشكل عام. لذلك علينا أن نبدأ أولا بعمل هذه الحقيبة ونضيفها إلى الكرة التي تمثل اللاعب. البريمج الخاص بهذه الحقيبة هو InventoryBox والموضح في السرد 29.

```
using UnityEngine;
using System.Collections.Generic;

public class InventoryBox : MonoBehaviour {

	//كم هو المبلغ الذي يملكه اللاعب حاليا
	public int money = 0;

	void Start () {

	}

	void Update () {

	}
}
```

السرد 29: حقيبة اللاعب

على الرغم من بساطة هذا البريمج حيث أنه يحتوي على متغير واحد فقط وهو المبلغ المالي money، إلا أنه يمكننا أن نضيف إليه ما نشاء من متغيرات لتخزين ما يمتلكه اللاعب من أشياء. هذا البريمج يعطي حامله القدرة على تجميع القطع النقدية، لذا يمكننا أن نختار من شخصيات اللعبة من يمكنه أن يجمعها ومن لا يمكنه بسهولة، وذلك عبر إضافة أو عدم إضافة هذا البريمج. فقد ترغب مثلا بجعل شخصيات الخصوم قادرة على التقاط الأسلحة والذخائر، ولكنها غير قادرة على تجميع الأموال. لنعد الآن للقطعة النقدية ونضيف إليها البريمج الخاص بمنطق عملية الجمع وهو Coin الموضح في السرد 30.

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour {

	//هذا المبلغ ستتم إضافته لرصيد حامل الحقيبة عندما
	//يقوم بجمع هذه القطعة النقدية
	public int coinValue = 1;

	void Start () {

```

```
	}

	void Update () {

	}

	//Collected هذه الدّالة تختص باستقبال الرسالة
	//Collectable والتي يتم إرسالها من قبل البريمج
	public void Collected(Collector owner){
		//علينا أولا التحقق من أن المجمّع يمتلك حقيبة لجمع النقود
		InventoryBox box = owner.GetComponent<InventoryBox>();
		if(box != null){
			//الحقيبة موجودة، لذا يمكننا أن نضيف
			//قيمة القطعة النقدية إلى الرصيد
			box.money += coinValue;

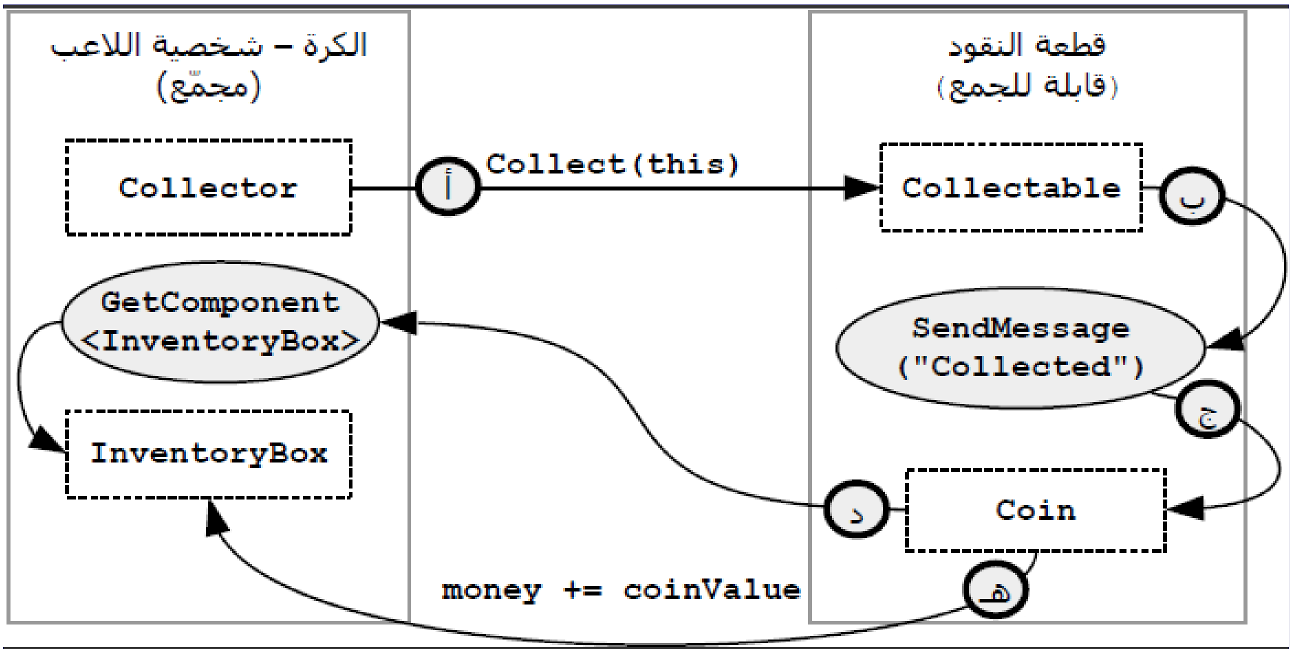
			//نقوم أخيرا بحذف القطعة من المشهد
			Destroy(gameObject);
		}
	}
}
```

السرد 30: البريمج الخاص بالقطع النقدية القابلة للجمع

كما تلاحظ فإن هذا البريمج أيضا لا يقوم بأي عمل بشكل منفرد (الدّالتان ()Start و ()Update خاليتان). المتغير الوحيد الذي فمنا بتعريفه هو coinValue والذي يسمح لنا بتحديد قيمة مختلفة لكل قطعة نقدية. الجزء الأهم في هذا البريمج هو الدّالة ()Collected والذي تأخذ المتغير owner من نوع Collector. بالعودة إلى السطر 23 في البريمج Collectable (السرد 26 في الصفحة 85) نتذكر أن هذا البريمج يرسل الرسالة Collected ويضمنها المرفق owner وهو الذي يقوم بمحاولة الجمع. هذه الدّالة تحمل نفس الاسم وتأخذ متغيرا من نفس نوع المرفق الذي يصاحب الرسالة وهو Collector، وهذه المميزات تجعلها قادرة على استقبال الرسالة Collected. باختصار، يمكن القول أنه من أجل أن تستقبل رسالة مرسلة عن طريق الدّالة ()SendMessae فما عليك سوى أن تعرف دالّة عند المستقبِل تحمل نفس اسم الرسالة، وإذا كانت الرسالة تحمل مرفقا، يجب أن تأخذ دالّة المستقبِل متغيرا بنفس نوع المرفق. أخيرا، عندما تصل الرسالة للمستقبِل فإنّ الدالّة التي تحمل اسم الرسالة سيتم تنفيذها.

لننتقل الآن إلى منطق الدّالة ()Collected وهو عمليا إضافة المبلغ coinValue إلى رصيد المجمّع owner. كما سبق وذكرت فإن جمع المال يعتمد على امتلاك المجمّع للحقيبة InventoryBox وهذا ما تقوم الدّالة بالتأكد منه. باستدعاء الدّالة ()<owner.GetComponent<InventoryBox فإننا نحاول العثور على مكوّن من نوع InventoryBox داخل الكائن الخاص بـ owner. إذا وُجِد هذا المكوّن فإننا نقوم بتخزين قيمته في المتغير box. إذا كانت قيمة box تساوي null فهذا يعني إن الحقيبة غير موجودة وبالتالي لا تقوم الدّالة بتنفيذ أي شيء. أمّا إذا كانت القيمة غير ذلك فهذا يعني أن الحقيبة InventoryBox موجودة وبالتالي نقوم بزيادة قيمة المتغير money بمقدار coinValue ومن ثم نقوم بحذف قطعة النقود من المشهد.

الشكل 36 يلخص بمخطط سهمي تفاصيل التفاعلات بين البريمجات Collector و Collectable و Coin و CategoryBox لإتمام عملية جمع النقود وإضافتها للحقيبة.



الشكل 36: تفاعلات عملية جمع قطعة النقود بين جميع البريمجات المعنية

يلخص المخطط في الشكل 36 التفاعلات بين البريمجات المختلفة الداخلة في عملية الجمع والتي تتم بعد اكتشاف التصادم بين المجمّع وقطعة النقود. في الخطوة أ يقوم المجمّع أي اللاعب باستدعاء الدّالة ()Collect من البريمج Collectable والموجود في قطعة النقود ممررا نفسه عبر المتغير this كمالك لما سيتم جمعه. في الخطوة ب يقوم البريمج Collectable بإرسال الرسالة Collected إلى جميع البريمجات الأخرى في قطعة النقود. هذه الرسالة يتم استقبالها في الخطوة ج من قبل البريمج Coin والذي يقوم بدوره في الخطوة د بفحص وجود الحقيبة (بريمج InventoryBox) لدى المجمّع المزوّد عبر المتغير owner. إذا وُجِد هذا البريمج ينقلنا Coin إلى الخطوة هـ والتي يتم خلالها زيادة الرصيد المالي في حقيبة اللاعب بمقدار قيمة القطعة النقدية الموجود في المتغير coinValue ومن ثم حذف قطعة النقود من المشهد.

نفس الخطوات المذكورة سيتم اتباعها فيما يتعلق بجمع الطعام، حيث سيكون الاختلاف الوحيد هو في التأثير على اللاعب. سيكون لدينا نوعان من الطعام: أخضر وأحمر. الاختلاف بينهما سيكون في مدة ومقدار التأثير على اللاعب حيث سيعمل كل منهما على زيادة حجم الكرة بمقدار مختلف ولمدة مختلفة. لأجل ذلك من الأفضل أن نقوم بعمل قالب خاص بكل نوع منهما. لنبدأ أولا مع البريمجات: كما أن بريمج النقود Coin يحتاج إلى بريمج الحقيبة InventoryBox لتتم عملية الجمع، فإن بريمج الطعام يحتاج إلى بريمج مقابل عند اللاعب لتتم عملية تناول الطعام وإحداث تأثيره. البريمج الأول الذي

سنتناوله هو البريمج المستقبِل للطعام من جهة اللاعب وهو SizeChanger الموضح في السرد 31 والذي يعمل على تغيير قياس الكرة بمقدار محدد ولمدة محددة. بعد ذلك سنتناول البريمج Food الموضح في السرد 32 الخاص بالطعام والذي سنضيفه إلى قالب الطعام الأحمر والأخضر جنبا إلى جنب مع Collectable و YRotator.

```
using UnityEngine;
using System.Collections;

public class SizeChanger : MonoBehaviour {

	//القياس الحالي للكائن
	float currentSize = 0;

	//مرجع للبريمج Collector المضاف إلى الكائن
	Collector col;

	void Start () {
		col = GetComponent<Collector>();
	}

	void Update () {

	}

	//يقوم البريمج Food باستدعاء هذه الدّالة محاولا إعطاء الطعام للمجمّع
	public bool IncreaseSize(float amount, float duration){
		//لا يمكن تغيير الحجم الحالي إلى لو كان صفرا
		if(currentSize == 0){
			//نقوم بتغيير الحجم الحالي إلى قيمة المتغير amount
			currentSize = amount;
			transform.localScale = Vector3.one * currentSize;
			//نقوم باستدعاء DecreaseSize() بعد انقضاء الوقت المحدد بـ duration
			Invoke("DecreaseSize", duration);

			//إذا وُجد البريمج Collector قم بزيادة قيمة radius
			if(col != null){
				col.radius = col.radius * currentSize;
			}
			//إرجاع القيمة true يعني أن الطعام تم تناوله
			return true;
		}
		//إرجاع القيمة false يعني أن الطعام لم يتم تناوله
		return false;
	}

	//تقوم هذه الدّالة بإرجاع الكائن لحجمه الأصلي
	public void DecreaseSize(){
		transform.localScale = Vector3.one;
		//نقوم باسترجاع القيمة الأصلية لـ radius في حال وجد البريمج Collector
		if(col != null){
			col.radius = col.radius / currentSize;
		}
```

```
            //قم بتغيير قيمة currentSize إلى صفر مرة أخرى
            currentSize = 0;
      }
}
```

السرد 31: البريمج الخاص بتناول الطعام وتغيير الحجم بشكل مؤقت

```
using UnityEngine;
using System.Collections;

public class Food : MonoBehaviour {

      //مقدار زيادة الحجم التي يسببها هذا الطعام
      public float sizeIncrementAmount = 2;

      //المدة الزمنية لتأثير الطعام مقدرة بالثواني
      public float incrementDuration = 5;

      void Start () {

      }

      void Update () {

      }

      //الهدف من تعريف هذه الدّالة هو استقبال الرسالة Collected
      //التي يرسلها Collectable وذلك حتى ننفذ عملية التجميع
      public void Collected(Collector owner){
            //يجب أن يحتوي المجمّع على البريمج SizeChanger حتى يتأثر بالطعام
            SizeChanger changer = owner.GetComponent<SizeChanger>();
            if(changer != null){
                  //البريمج موجود، فنحاول أن نعطي الطعام للمجمّع
                  bool canTake =
                        changer.IncreaseSize(
                              sizeIncrementAmount, incrementDuration);
                  //هل قام المجمّع بإخذ الطعام؟
                  if(canTake){
                        //نعم، إذن علينا في هذه الحالة أن نحذف الطعام من المشهد
                        Destroy(gameObject);
                  }
            }
      }
}
```

السرد 32: البريمج الخاص بالطعام القابل للجمع

البريمج الأول SizeChanger يحتوي على دالّتين رئيسيتين هما ()IncreaseSize و ()DecreaseSize. بالإضافة لذلك يحتوي على المتغير col والذي نحاول في بداية تشغيل البريمج أن نخزن فيه مرجعا للبريمج الخاص بالجمع Collector إن وجد. عندما يلمس المجمّع كائنا قابلا للجمع من نوع Food أي طعام يتم استدعاء الدّالة ()IncreaseSize وتزويدها بمتغيرين هما مقدار الزيادة في الحجم amount بالإضافة إلى مدة التأثير duration. ما تقوم به هذه الدّالة هو التحقق أولا من أن currentSize تساوي

صفرا، مما يعني أن الكائن حاليا في حجمه الطبيعي وليس تحت تأثير طعام سابق تم جمعه. بعد التحقق من هذا الشرط يتم تغيير currentSize إلى قيمة amount وتغيير قياس الكائن بمقدار amount أيضا. بعد ذلك نستخدم الدّالة ()Invoke لاستدعاء الدّالة ()DecreaseSize لإعادة الحجم الأصلي. الميزة التي تعطينا إياها ()Invoke هي إمكانية تأجيل استدعاء الدّالة التي نرغب بها لمدة زمنية محددة. في هذه الحالة نريد أن يعود الكائن لحجمه الأصلي بعد زوال مدة تأثير الطعام والمحددة عن طريق المتغير duration. أي أننا عندما نستدعي ()Invoke كما في السطر 28 فإننا نخبر البريمج SizeChange بأن يقوم باستدعاء ()DecreaseSize لكن بعد مرور duration من الثواني.

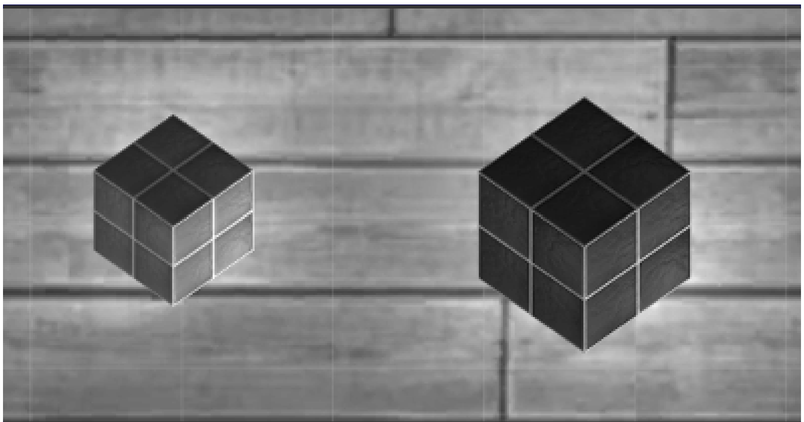
كل ما تم من تغييرات حتى الآن لا يؤثر في إمكانية البريمج Collector على الجمع وذلك لأن عملية اكتشاف التصادمات تعتمد على المتغير radius الذي لا زال على حاله. من أجل ذلك نقوم في السطر 32 بتغيير قيمة radius عن طريق ضربها بقيمة amount بالتالي نزيد من نصف قطر التصادم الخاص بالمجمّع مما يجعله قادرا على التقاط الأشياء من مسافة أبعد بالتالي تسهيل وتسريع عملية الجمع. من المهم هنا الإشارة إلى أن استدعاء ()IncreaseSize لا يعني بالضرورة أن الطعام تم أخذه وأن تأثيره حدث، فالبريمج قد يكون أصلا تحت تأثير طعام آخر مما يمنعه من أخذ غيره وبالتالي فإن القيمة التي سترجعها ()IncreaseSize هي false، أما إذا تمت عملية تناول الطعام وتغيير الحجم فإن القيمة التي سترجع هي true.

بعد انقضاء المدة المحددة لتأثير الطعام يتم تنفيذ ()DecreaseSize والتي تعمل على استرجاع القياس الأصلي للكائن بالإضافة إلى إعادة currentSize إلى القيمة صفر. طبعا يجب ألا ننسى قبل ذلك إعادة قيمة نصف القطر radius الخاصة بـ Collector إلى مقدارها الأصلي عن طريق قسمتها على currentSize.

على الطرف الآخر من عملية تناول الطعام يوجد البريمج Food والذي يشبه البريمج Coin في كنه يستقبل الرسالة Collected ويقوم بترجمتها لشيء ذي معنى. هنا نتذكر أن البريمج Coin اعتمد على وجود الحقيبة InventoryBox على الطرف الآخر، وبالمثل فإن Food يعتمد على وجود SizeChanger. لذا فالخطوة الأولى التي يقوم بها البريمج Food عند استقبال الرسالة Collected هي التأكد من وجود SizeChanger لدى المجمّع المفترض owner. إذا وُجد هذا البريمج فإنه يتم استدعاء الدّالة () IncreaseSize الخاصة به وتزويدها بقيمة كل من sizeIncrementAmount و incrementDuration. لاحظ أن هذين المتغيرين هما public مما يمكننا لاحقا من تحديد قيمتهما من خلال نافذة الخصائص. عند استدعاء ()IncreaseSize فإنها سترجع قيمة من نوع bool لذا نقوم بتخزينها في المتغير canTake. إذا كانت قيمة canTake هي true فهذا يعني أن الطعام قد تم أخذه وبالتالي يمكننا أن نقوم بحذفه من المشهد. أما إذا كانت القيمة false فهذا يعني أن المجمّع لم يتمكن من أخذ الطعام في هذه المحاولة، بالتالي يبقى الطعام في المشهد ولا شيء يتغير.

أخيرا يمكننا أن نقوم بعمل قوالب خاصة بكل من الطعام الأخضر والأحمر وذلك باستخدام مكعبات كما

في الشكل 37. بعد ذلك نضيف البريمجات YRotator و Collectable و Food إلى كل من هذه القوالب. لتغيير تأثير أنواع الطعام المختلفة يمكننا ببساطة أن نغير كلا من القيم sizeIncrementAmount و incrementDuration. فمثلا يمكننا إعداد الطعام الأخضر ليزيد الحجم بمقدار 2 لمدة سبع ثوان ونصف والطعام الأحمر ليزيد الحجم بمقدار 3.5 لمدة 5 ثوان. يمكنك الاطلاع على النتيجة النهائية في المشهد scene10 في المشروع المرفق.



الشكل 37: قوالب الطعام المستخدمة. الأحمر إلى اليمين والأخضر إلى اليسار

الفصل الثالث: إمساك وإفلات الأشياء

من المهم أحيانا إعطاء اللاعب القدرة على تحريك العناصر في المشهد، بحيث يمكنه – مثلا – تكديس بعض الصناديق لتسلقها أو إزالة بعض العقبات من الطريق وهلم جرا. هذا التحريك قد يتم بعدة أشكال، قد تكون بسيطة مثل دفع العناصر بمجرد التحرك باتجاهها، أو قد تكون عن طريق قوى خارقة أو أدوات خاصة مثل بندقية الجاذبية في لعبة Half-Life 2 المعروفة. في هذا الفصل سنعمل على استثمار العلاقات بين الكائنات بغرض بناء آلية لحمل الأشياء، بحيث يقوم اللاعب بحمل العنصر والمشي به ثم إفلاته في مكان آخر.

سنقوم في هذا الفصل بإعادة استعمال نظام إدخال منظور الشخص الأول الذي سبق وقمنا ببنائه في الفصل الرابع من الوحدة الثانية، ولذا يمكننا أن نبدأ العمل انطلاقا من المشهد scene5 في المشروع المرفق. التعديل البسيط الذي سنجريه على هذا النظام هو إعطاء اللاعب القدرة على حمل الصناديق عن طريق الضغط على مفتاح E وإفلاتها باستخدام نفس المفتاح، ومن أجل ذلك سنحتاج لإضافة بريمجين جديدين. البريمج الأول هو Holdable الموضح في السرد 33 والذي سنضيفه على كافة العناصر التي نريد أن نعطي اللاعب القدرة على حملها. في واقع الأمر سيكون هذا بريمجا فارغا تقريبا حيث لا يحتوي إلا على متغير واحد وهو نصف القطر radius والذي سنستخدمه لتحديد ما إذا كان اللاعب قريبا كفاية من الكائن ليقوم بحمله. وتذكر دائما أنه من الأفضل عمل قالب خاص بالصندوق

القابل للحمل حيث أننا سنضيف عددا من هذه الصناديق للمشهد.

```
using UnityEngine;
using System.Collections;

public class Holdable : MonoBehaviour {

      //نصف قطر الكائن القابل للحمل
      public float radius = 1.5f;

      void Start () {

      }

      void Update () {

      }
}
```

السرد 33: البريمج الخاص بالكائنات القابلة للحمل

الجزء الأكبر من العمل سيقوم بإنجازه البريمج Holder والذي يجب أن نضيفه لكائن اللاعب (الأسطوانة). هذا البريمج موضح في السرد 34.

```
using UnityEngine;
using System.Collections;

public class Holder : MonoBehaviour {

      //نصف قطر كائن الحمل
      public float radius = 0.5f;

      Holdable objectInHand;

      void Start () {

      }

      void Update () {
            if(Input.GetKeyDown(KeyCode.E)){
                  //إن لم يوجد في أيدينا كائن سلفا
                  //علينا أن نبحث عن كائن مناسب لحمله
                  if(objectInHand == null){
                        //البحث عن كافة الكائنات التي يمكن حملها
                        Holdable[] allHoldables =
                              FindObjectsOfType<Holdable>();
                        foreach(Holdable holdable in allHoldables){
                              //حساب المسافة بين الكائن المحمول
                              //واللاعب الذي يحاول حمله
                              float distance =
                                    Vector3.Distance(
                                                transform.position,
```

```
29.                                         holdable.transform.position);
30.
31.                             //أولا يجب أن تكون المسافة قصيرة كفاية
32.                             bool close =
33.                                   distance < radius + holdable.radius;
34.
35.                             //ثانيا يجب أن يكون اللاعب مواجها للكائن المراد حمله
36.                             Vector3 dVector;
37.                             dVector = holdable.transform.position
38.                                                 - transform.position;
39.
40.                             float ang =
41.                                   Vector3.Angle(dVector,
42.                                         transform.forward);
43.
44.                             if(close && ang < 90){
45.                                   //يمكننا الآن حمل الكائن
46.                                   //holdable أولا نقوم بتخزين قيمته في
47.                                   objectInHand = holdable;
48.
49.                                   //ثانيا نقوم بإضافة الكائن المحمول كابن
50.                                   //لكائن اللاعب الذي يحمله
51.                                  holdable.transform.parent = transform;
52.
53.                                   return;
54.                             }
55.
56.                       }
57.                 } else {
58.                       //يوجد بين أيدينا كائن سلفا
59.                       //لذا علينا الآن أن نقوم بإفلات هذا الكائن
60.                       objectInHand.transform.parent = null;
61.                       objectInHand = null;
62.                 }
63.           }
64.     }
65. }
```

السرد 34: بريمج الحمل الخاص باللاعب

الفكرة وراء هذا البريمج بسيطة وتتلخص في أنه عندما يضغط اللاعب على المفتاح E فإنه يقوم بالتحقق من وجود كائن حاليا في يدي اللاعب. فإذا لم يكن هناك كائن يتم البحث عن كائن مناسب ليتم حمله. وتتم عملية الإمساك والحمل فعليا إذا وُجِد هذا الكائن المناسب. أما إذا كان هناك كائن موجود سلفا بين يدي اللاعب فإن هذا الكائن سيتم إفلاته. هذه المعلومة يمكن معرفتها عن طريق قيمة المتغير objectInHand والذي يخزن قيمة الكائن الذي يحمله اللاعب حاليا، وبالطبع فإن القيمة null تعني أن اللاعب لا يحمل أي شيء كما نلاحظ في السطر 19.

عملية إمساك وحمل الكائنات تتم عبر عدة مراحل، وتبدأ بالبحث عن جميع كائنات المشهد التي تحمل البريمج Holdable ومن ثم المرور عبرها جميعا كما في الأسطر من 21 إلى 23. خلال عملية المرور هذه

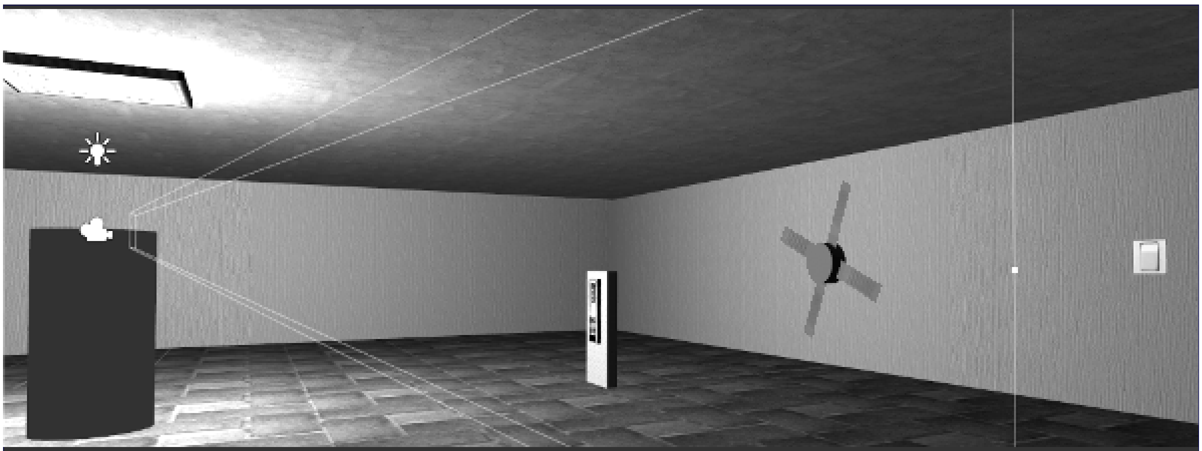
نقوم بمقارنة المسافة بين الكائن واللاعب ومقارنتها بمجموع أنصاف الأقطار للكائنين. إذا تحقق هذا الشطر فإننا نقوم بتغيير قيمة close إلى true كما في الأسطر 26 إلى 33. إلاّ أن شرط قرب المسافة وحده لا يكفي إذ لابد أن يكون اللاعب مواجها للصندوق أو الكائن الذي يريد أن يحمله، وهذا يتم عن طريق قياس الزاوية اتجاه نظر اللاعب transform.forward والخط المستقيم الواصل بين موقع اللاعب وموقع الكائن الذي يحاول أن يحمله، وذلك في الأسطر من 36 إلى 42. بالدخول قليلا إلى حساب المتجهات، نتعلم أنه يمكننا الحصول على المتجه الواصل بين موقعي الكائن واللاعب عن طريق طرح الثاني من الأول. إذا كانت الزاوية التي قمنا بحسابها أقل من 90 درجة فإننا نعتبر بذلك أن اللاعب يواجه الصندوق وبالتالي يمكنه حمله كما في السطر 44.

بعد التحقق من كافّة الشروط الواجب توفرها يمكن أن تتم عملية الإمساك بالصندوق، وهي عملية بسيطة جدا لا تعدو تخزين قيمة الكائن الذي تم حمله في المتغير objectInHands، مما يجعل التأثير المقبل للمفتاح E هو إفلات الكائن. بالإضافة لذلك علينا أن نضيف الكائن المحمول كابن لكائن اللاعب وذلك حتى يتحرك ويستدير معه وهو المغزى من عملية الإمساك كما ترى في الأسطر 44 إلى 51. لاحظ أننا في السطر 53 نقوم باستخدام return من أجل إيقاف تنفيذ الدّالة ()Update بعد أن حققنا مبتغانا في حمل الكائن، وهذا يؤدي إلى تحسين الأداء من جهة بمنع أي مقارنات غير ضرورية كما أنه يمنع حمل أكثر من كائن في المرة الواحدة. أخيرا نلاحظ الأسطر 57 إلى 62 والتي تنطبق في حال الضغط على مفتاح E أثناء حمل كائن ما، في هذه الحالة يجب أن نقوم بإفلات الكائن وذلك بإزالته من أبناء اللاعب عن طريق تغيير قيمة transform.parent الخاصة به إلى null، ولا ننسى أيضا إعادة قيمة objectInHand إلى null من أجل تحرير يدي اللاعب ليتمكن لاحقا من الإمساك بكائن آخر. يمكنك مشاهدة النتيجة النهائية في المشهد scene11 في المشروع المرفق.

الفصل الرابع: المحفّزات ومفاتيح التحكم

تحريك الأشياء ليس – بطبيعة الحال – الطريقة الوحيدة التي يمكن للاعب من خلالها التأثير في المشهد، بل يمكنه أيضا أن يقوم بتفعيل أو تعطيل بعض العناصر أو الآلات كالمصابيح الكهربائية مثلا. عملية التفعيل أو التعطيل هذه تعرف بالـ"التحفيز"، ذلك أن اللاعب يقوم بحدث ما يحفّز حدثا آخر. فمثلا عندما يقوم اللاعب بإشعال أو إطفاء المصباح الكهربائي، فإنه في الواقع يتعامل مع المفتاح الذي يتحكم بهذا المصباح، وهذا المفتاح بدوره يقوم بالعمل المطلوب. في هذه الحالة نقول بإنّ المفتاح هو المحفّز لأنه من قام بالعمل الفعلي، أما اللاعب فهو العنصر الذي قام بتنشيط هذا المحفّز ليقوم بدوره.

سنطلع في الفصل إن شاء الله على نظام عام يمكن تطبيقه على أنواع مختلفة من المحفّزات، مما سيجعل الكود الذي سنكتبه يبدو معقدا نوعا ما للوهلة الأولى، إلا أنه على المدى البعيد يعطيك حلا سلسا ونمطا يمكنك تكرار استخدامه بمرونة في معظم المواقف التي تحتاج فيها إلى أفعال ومحفّزات. إضافة لذلك فإن المشهد الذي سنتعامل معه في هذا الفصل سيكون أيضا أكثر تعقيدا مما سبقه وسيحتوي على عدد لا بأس به من العناصر. لنبدأ أولا بالصورة العامّة للمشهد كما تظهر في الشكل 38.



الشكل 38: المشهد الذي سنستعمله لشرح استخدام المحفّزات ومفاتيح التحكم

لنبدأ بتعداد مكونات المشهد التي نراها في الشكل 38 من اليسار لليمين. لدينا أولا أسطوانة تمثل نظام تحكم منظور الشخص الأول. بالنسبة لعناصر المشهد لدينا أولا ضوء نقطي سنستخدمه لتمثيل المصباح الكهربائي، وهناك أيضا لوحة تحكم منصوبة بشكل عمودي سنستخدمها للتحكم بالمروحة الكبيرة على الجدار. أخيرا لدينا في أقصى اليمين مفتاح كهربائي سيستعمل للتحكم في المصباح الكهربائي. لو أردنا أن نصنّف هذه المكونات يمكننا أن نضع المروحة والمصباح الكهربائي في خانة الكائنات القابلة للاستعمال، بينما تندرج لوحة التحكم بالمروحة والمفتاح الكهربائي في فئة المحفّزات.

لنبدأ أولا مع البريمج الخاص بالمحفّزات، وهي التي يمكن للاعب استخدامها لإحداث تغييرات في المشهد وتفعيل أو تعطيل بعض العناصر. سنسمي هذا البريمج SwitchableTrigger وهو موضح في السرد 35 للدلالة على أنه يمكن للاعب استخدامه كمفتاح أو لوحة تحكم، حيث هناك أنواع أخرى من المحفّزات منها على سبيل المثال محفّزات ترتبط بالوقت فتعمل مثلا بعد 5 ثوان من حدث ما أو ترتبط بلمس اللاعب لكائن معين.

```
using UnityEngine;
using System.Collections;

public class SwitchableTrigger : MonoBehaviour {

        //يتنقل المحفّز بين هذه الحالات المختلفة مع كل استخدام
        public TriggerState[] states;

        //الموقع الخاص بالحالة التي عليها المحفّز في الوقت الراهن
        public int currentState = 0;

        //أبعد مسافة يمكن ابتداء منها التعامل مع هذا المحفّز
        public float activationDistance = 3;

        //آخر وقت تم فيه تغيير حالة المحفّز
        float lastSwitchTime = 0;
```

```
    void Start () {

    }

    void Update () {

    }

    //تحاول هذه الدّالة تغيير حالة المحفّز والانتقال إلى الحالة التالية
    //إذا نجحت محاولة الانتقال فإنها تعيد القيمة true
    public bool SwitchState(){
        //إذا كانت مصفوفة الحالات فارغة فليس هناك ما يمكن فعله
        if(states.Length == 0){
            return false;
        }

        //قم باستدعاء الحالة التي نحن عليها في الوقت الراهن
        TriggerState current = states[currentState];

        //تأكد من أن وقت الانتظار الخاص بهذه الحالة قد انقضى
        if(Time.time - lastSwitchTime > current.restTime){
            //إذا انقضى ذلك الوقت فعلا، حينها يمكننا التبديل إلى حالة جديدة
            currentState += 1;

            //إذا كنا في الحالة الأخيرة فإننا نرجع إلى الأولى
            if(currentState == states.Length){
                currentState = 0;
            }

            //احصل على الحالة الجديدة التي نرغب بالتحول إليها
            TriggerState newState = states[currentState];

            //قم بإرسال جميع الرسائل المخزنة في الحالة الجديدة
            foreach(TriggerMessage message
                                    in newState.messagesToSend){
                //احصل على مستقبل الرسالة
                GameObject sendTo = message.messageReceiver;
                //احصل على اسم الرسالة
                string messageName = message.messageName;
                //قم بإرسال الرسالة إلى مستقبلها
                sendTo.SendMessage(messageName);
            }

            //نقوم أخيرا بتسجيل وقت التبديل إلى الحالة الجديدة
            lastSwitchTime = Time.time;
            return true;
        } else {
            return false;
        }
    }
}

//بنية صغيرة تستخدم لتخزين الرسائل
[System.Serializable]
```

```
72. public class TriggerState{
73.      public float restTime;
74.      public TriggerMessage[] messagesToSend;
75. }
76.
77. //بنية أخرى تمثل الرسائل التي يمكننا إرسالها
78. [System.Serializable]
79. public class TriggerMessage{
80.      public GameObject messageReceiver;
81.      public string messageName;
82. }
```

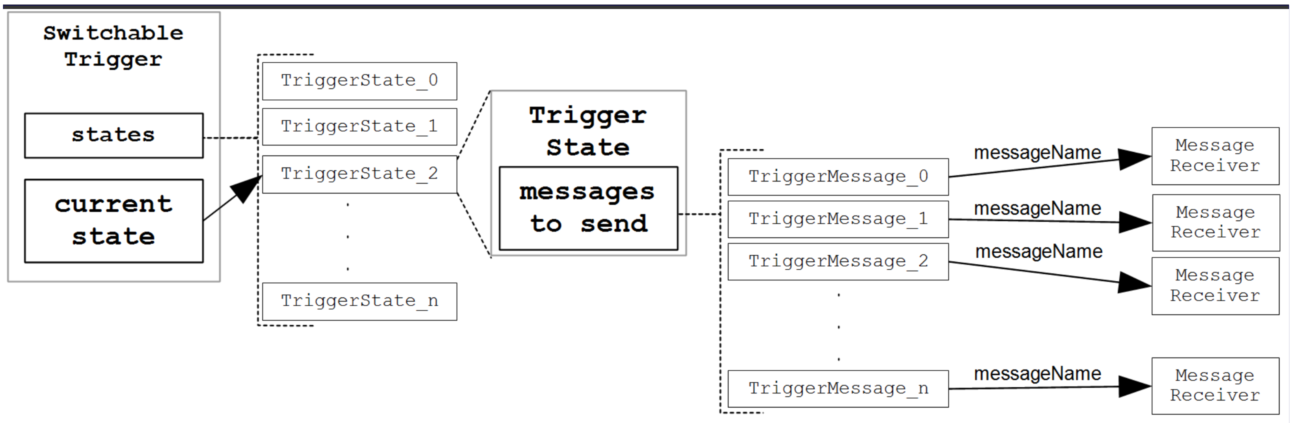
السرد 35: البريمج الخاص بالمحفّزات التي يمكن للاعب تفعيلها يدويا

قبل الدخول في تفاصيل البريمج نفسه، أود الانتقال أولا إلى آخر السرد تحديدا الأسطر 72 إلى 76 و 79 إلى 83. تلاحظ في هذه الأسطر أننا قمنا بتعريف وحدات برمجية تختلف عن البريمجات التي عهدناها، هذه الوحدات تعرف بالبُنى البرمجية structures. بالرغم من أن تعريفها شبيه جدا بتعريف البريمجات إلى أنها تختلف من ناحية أنها لا ترث من MonoBehavior، كما أن تعريفها مسبوق بالأمر [System.Serializable]. هذه الوحدات البرمجية بسيطة التركيب سنستخدمها كأنها متغيرات نخزن فيها بعض القيم لا أكثر، فهي بالتالي لا تحتوي على أية دوالّ برمجية يمكن استدعاؤها ولا تملك بنفسها أي منطق برمجي قابل للتنفيذ. فإذا عرفنا – مثلا – متغيرا من نوع TriggerState فإن هذا المتغير يحتوي في داخله على متغيرين آخرين هما restTime والمصفوفة messagesToSend. أما أهمية الأمر [System.Serializable] فهي جعل هذا المتغير ظاهرا وقابلا للتعديل والتحرير في نافذة الخصائص داخل Unity.

سنستخدم النوع TriggerState لتعريف الحالات المختلفة التي يمكن للمحفّز أن يكون عليها، فمثلا بالنسبة لمفتاح المصباح الكهربائي هناك حالتان فقط هما "مشغّل" و"مطفأ". لكن هذا لا يمنع وجود أكثر من حالة في أمثلة أخرى. لكل حالة من الحالات هناك وقت انتظار خاص بها وهو restTime والذي يجب أن ينقضى كاملا قبل أن نسمح للمحفز بالانتقال من حالته التي هو عليها إلى حالة أخرى. هذا الوقت قد يكون مفيدا في الحالات التي تتطلب وقتا مثل تحريك مصعد أو فتح باب كهربائي، مما يلزمنا بقفل المحفّز مؤقتا ريثما تتم العملية المطلوبة. لكل حالة من الحالات هناك مصفوفة تسمى messagesToSend وتحتوي على عناصر من نوع TriggerMessage. بالاطلاع على TriggerMessage نجد أن كل متغير من هذا النوع سيحتوي على كائن messageReceiver وهو الذي سيقوم باستقبال الرسالة و messageName وهو اسم الرسالة التي سنرسلها له. بكلمات أخرى فإن messageReceiver يجب أن يحتوي على الأقل على بريمج واحد يمكنه استقبال رسالة تحمل الاسم المخزن في messageName. لا تقلق إذا بدا الأمر إلى الآن معقدا، فبالمثال ستتضح الصورة أكثر إن شاء الله.

في كل مرة يحاول فيها اللاعب استعمال المحفّز، فإنه عمليا يقوم باستدعاء الدّالة ()SwitchState والتي بدورها تقوم بإبلاغ اللاعب بنجاح أو فشل محاولة التفعيل التي قام بها وذلك عن طريق إرجاع true أو false. أحد الأسباب التي قد تؤدي لفشل محاولة التفعيل هو عدم انقضاء وقت الانتظار

المطلوب للحالة التي عليها المحفّز الآن. في حال نجاح التفعيل فإن قيمة currentState تزداد بمقدار واحد، أو تعود للقيمة صفر (الحالة الأولى) في حال كانت أصلا على آخر حالة ممكنة. بعد التغيير إلى الحالة الجديدة يقوم المحفز بالمرور على كافّة عناصر المصفوفة messagesToSend والمخزنة في الحالة الجديدة، ومن ثم يقوم بإرسال كل رسالة إلى مستقبلها كما في الأسطر 51 إلى 59. قبل أن نبلغ اللاعب بنجاح التفعيل ونعيد له القيمة true علينا أولا أن نقوم بتسجيل الوقت الذي تمت فيه عملية التفعيل بحيث نتمكن من حساب وقت الانتظار في محاولة التفعيل المقبلة. الشكل 39 يوضح الآلية التي يتم فيها إحداث عدد من التغييرات المتزامنة في المشهد من خلال تفعيل محفّز واحد.



الشكل 39: آلية التحفيز: عندما تتغير حالة المحفّز يتم إرسال مجموعة من الرسائل المخزنة في المصفوفة messagesToSend الخاصة بالحالة الجديدة

لننتقل الآن إلى الأمثلة العملية التي سترينا كيف تعمل كل هذه الأشياء معا. لنبدأ بالمثال الأسهل وهو التحكم بالمصباح الكهربائي: بداية علينا أن نعرف أن كلا من المفتاح والمصباح له حالتان فقط هما التشغيل والإطفاء، فعندما يقوم اللاعب بتفعيل المحفّز وهو هنا المفتاح الكهربائي، سيتم إرسال رسالتين إحداهما للمصباح لتغيير حالته من التشغيل للإطفاء أو العكس، والأخرى للمفتاح نفسه ليتحرك نحو الأعلى أو الأسفل حسب الحالة. معنى هذا أنه لدينا حالتان لكل من المفتاح والمصباح، وعند انتقال المحفّز من حالة لأخرى فإنه يرسل رسالتين إحداهما للمفتاح والأخرى للمصباح لتغيير حالتيهما. لننتقل الآن للبريمجات التي ستستقبل هذه الرسائل من المحفّز، وأولها هو بريمج المصباح LightControl المبين في السرد 36. يمكن لهذا البريمج أن يستقبل الرسالتين SwitchOn و SwitchOff.

```
using UnityEngine;
using System.Collections;

public class LightControl : MonoBehaviour {

        //مكوّن الضوء الموجود على الكائن والذي ستتحكم به
        Light toControl;

        void Start () {
```

```
            toControl = GetComponent<Light>();
      }

      void Update () {

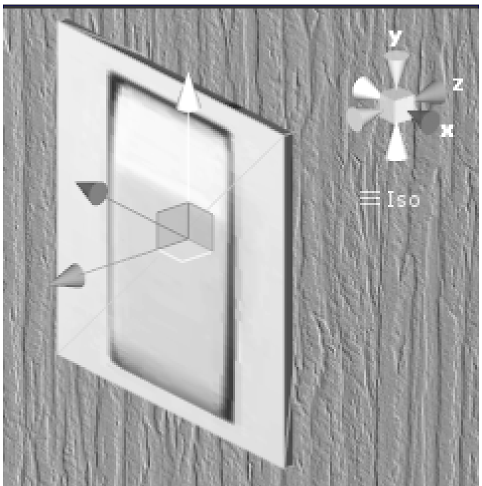
      }

      //"SwitchOn" دالّة خاصة باستقبال الرسالة
      public void SwitchOn(){
            toControl.enabled = true;
      }

      //"SwitchOff" دالّة خاصة باستقبال الرسالة
      public void SwitchOff(){
            toControl.enabled = false;
      }
}
```

السرد 36: بريمج يتحكم بالضوء عن طريق استقبال رسائل من المحفّز

هذا البريمج مصمم لتتم إضافته لكائن الضوء النقطي الذي سنستخدمه كمصباح كهربائي، ذلك أنه عند بداية التشغيل يقوم بالبحث عن مكوّن من نوع Light وهو لا يوجد إلّا على كائنات الأضواء، ومن ثم يقوم بتخزين هذا المكوّن في المتغير toControl. يقوم هذا البريمج ببساطة باستقبال الرسالة SwitchOn وتفعيل مكوّن الضوء بناء عليها، كما أنّه يقوم بتعطيل مكوّن الضوء بناء على استقبال الرسالة SwitchOff. البريمج الآخر والذي سنضيفه على كائن المفتاح هو ZFlipper الموضح في السرد 37، هذا البريمج يقوم بتدوير الكائن حول محوره المحلي z بمقدار 180 درجة عند استقباله للرسالة Flip. الهدف من هذا البريمج هو محاكاة حركة المفتاح نحو الأعلى نحو الأسفل عن طريق تدويره مما يغير اتجاه الإكساء. تأمل الشكل 40 والذي يمثل صورة مقربة للمفتاح، لو تم تدوير هذا الكائن بمقدار 180 درجة سيبدو وكأن المفتاح قد تم تحريكه نحو الأسفل.



الشكل 40: كائن المفتاح المستخدم للتحكم بالمصباح الكهربائي

```
using UnityEngine;
using System.Collections;

public class ZFlipper : MonoBehaviour {

      void Start () {

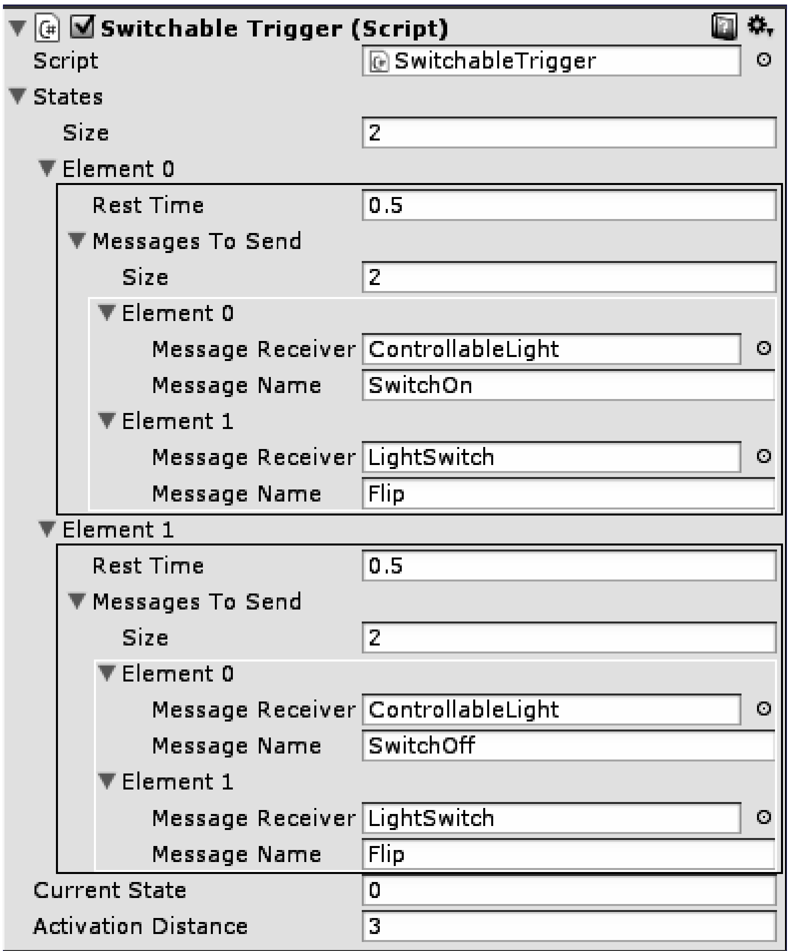
      }

      void Update () {

      }
      //تقوم هذه الدّالة بتدوير الكائن حول محوره المحلي z بمقدار 180 درجة
      public void Flip(){
            transform.Rotate(0, 0, 180);
      }
}
```

السرد 37: بريمج تدوير المفتاح الكهربائي

لدينا الآن إذن بريمج المحفّز القادر على إرسال رسائل مختلفة، كما لدينا أيضا بريمجان قادران على استقبل الرسائل وتنفيذ المهام المطلوبة بناء عليها. كل ما علينا الآن هو أن نضيف البريمجين ZFlipper و SwitchableTrigger إلى كائن المفتاح الكهربائي والبريمج LightControl إلى المصباح ومن ثم ربطها عن طريق نافذة الخصائص. الشكل 41 يوضح نافذة الخصائص وهي تعرض كائن المفتاح بعد إعداده بشكل كامل لتنفيذ المهمة بشكل صحيح.



الشكل 41: بريمج المحفّز بعد إعداده للتحكم في المصباح. الإطاران الأسودان يمثلان الحالتين اللتين ينتقل بينهما المحفّز، بينما يمثل الإطاران الأبيضان مجموعتي الرسائل التي يتم إرسالها إبّان تفعيل كل حالة

بدراسة الشكل 41 نلاحظ أن المحفّز ينتقل بين حالتين، حيث تقوم الحالة الأولى بإرسال الرسالة SwitchOn إلى كائن الضوء وهو هنا يحمل الاسم ControllableLight، بينما تقوم الحالة الثانية بإرسال الرسالة SwitchOff إلى نفس الكائن. لاحظ أيضا أن كلا الحالتين ترسلان الرسالة Flip إلى الكائن LightSwitch وهو نفس كائن المفتاح المضاف إليه البريمج SwithcableTrigger. بكلمات أخرى فإن البريمج يرسل الرسالة Flip إلى نفسه عن طريق SwitchableTrigger ومن ثم يقوم باستقبالها عن طريق ZFlipper. بقي أن نعطي اللاعب القدرة على تفعيل المحفّزات عن طريق الضغط على المفتاح E، لذا علينا أن نضيف البريمج TriggerSwitcher إلى الأسطوانة التي تمثل اللاعب. هذا البريمج موضح في السرد 38.

```
using UnityEngine;
using System.Collections;

public class TriggerSwitcher : MonoBehaviour {
```

```

    void Start () {

    }

    void Update () {
        if(Input.GetKeyDown(KeyCode.E)){
            //ابحث عن جميع المحفّزات الموجودة في المشهد
            SwitchableTrigger[] allST =
                FindObjectsOfType<SwitchableTrigger>();

            //البحث عن محفّز ملائم لتفعيله
            //المحفّز الملائم يجب أن يكون قريبا ومواجها للاعب
            foreach(SwitchableTrigger st in allST){
                float dist =
                    Vector3.Distance(transform.position,
                                    st.transform.position);

                //activationDistance إذا كانت المسافة أقل من
                //فهذا يعني أن المحفّز قريب من اللاعبة بشكل كاف لتفعيله
                if(dist < st.activationDistance){
                    Vector3 distVector =
                        st.transform.position
                            - transform.position;

                    float angle =
                        Vector3.Angle(distVector,
                                    transform.forward);
                    //إذا كانت الزاوية أقل من 90، فهذا يعني أن المحفّز مواجه للاعب
                    if(angle < 90){
                        //بما أنه مواجه يمكننا أن نحاول تفعيله
                        st.SwitchState();
                    }
                }
            }
        }
    }
}
```

السرد 38: البريمج الذي يمكن اللاعب من تفعيل المحفّزات باستخدام المفتاح E

عندما يضغط اللاعب على المفتاح E فإن البريمج يقوم بالبحث عن كافّة المحفّزات الموجودة في المشهد، ومن ثم يقوم بحساب المسافة بين اللاعب وكل من هذه المحفّزات. فإذا وجد أن المسافة أقل من قيمة activationDistance الخاصّة بالمحفّز، ينتقل إلى الخطوة التالية من التحقق وهي حساب الزاوية بين اتجاه نظر اللاعب ومتجه المسافة بينه وبين كائن المحفّز. هذه الخطوة تشبه ما قمنا بعمله في الفصل الثالث من هذه الوحدة عند برمجتنا لإمساك وإفلات الكائنات. فإذا تحقق هذان الشرطان يمكن حينها للبريمج أن يقوم باستدعاء الدّالة SwitchState() من المحفّز في محاولة لتفعيله. بعد أن تضيف البريمج TriggerSwitcher إلى كائن الأسطوانة تصبح آلية تشغيل وإطفاء المصباح جاهزة وبإمكانك تجربتها، كما يمكنك الاطلاع على النتيجة في المشهد scene12 في المشروع المرفق.

لنقم الآن بالعمل على مثال آخر من أجل تعزيز فهم الفكرة. تذكر أنك رأيت في الشكل 38 مروحة على الجدار ولوحة تحكم خاصة بها. سنعمل على جعل لوحة التحكم هذه مرتبطة بالمروحة بحيث تعمل على تشغيلها وإيقافها وتغيير سرعتها. البريمج ControllableFan الموضح في السرد 39 يقوم باستقبال رسائل التشغيل والإيقاف وتغيير السرعة وتطبيقها على المروحة.

```
using UnityEngine;
using System.Collections;

public class ControllableFan : MonoBehaviour {

	public float speed1 = 20;
	public float speed2 = 40;
	public float speed3 = 60;

	float currentSpeed = 0;

	void Start () {

	}

	void Update () {
		transform.Rotate(0, currentSpeed * Time.deltaTime, 0);
	}

	public void SetSpeed1(){
		currentSpeed = speed1;
	}

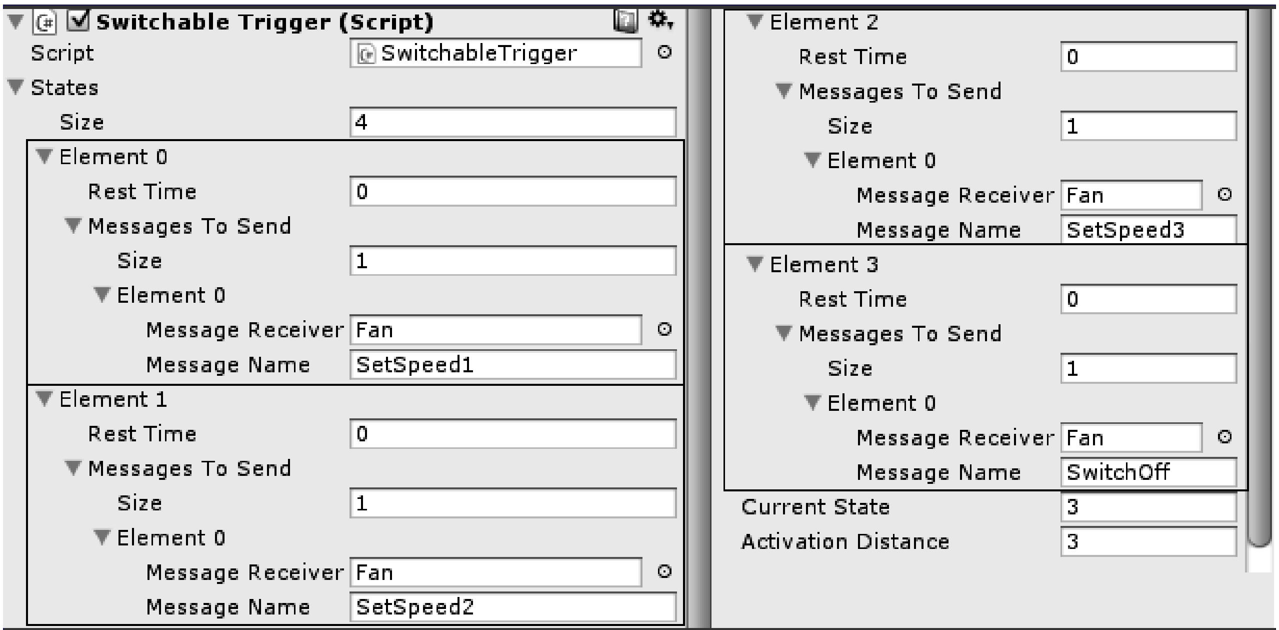
	public void SetSpeed2(){
		currentSpeed = speed2;
	}

	public void SetSpeed3(){
		currentSpeed = speed3;
	}

	public void SwitchOff(){
		currentSpeed = 0;
	}
}
```

السرد 39: بريمج المروحة

هذا البريمج يعمل ببساطة شديدة على التحكم بقيمة المتغير currentSpeed وبالتالي سرعة دوران المروحة وذلك عن طريق استقبال الرسائل المختلفة مثل SetSpeed1 و SetSpeed2. إضافة إلى ذلك يعمل البريمج على إيقاف المروحة وذلك بتغيير السرعة إلى صفر عند استقبال الرسالة SwitchOff. بعد إضافة هذا البريمج للمروحة علينا أن نقوم بإضافة SwitchableTrigger إلى لوحة التحكم الموجودة في منتصف الغرفة وإعداده وفقا الشكل 42.



الشكل 42: تهيئة المحفّز الخاص بالتحكم بالمروحة

هذه المرة لدينا 4 حالات مختلفة: 3 سرعات وحالة الإيقاف. لاحظ أن الحالات الثلاث الأولى ترسل الرسائل SetSpeed1 و SetSpeed2 و SetSpeed3 على التوالي، بينما ترسل الحالة الأخيرة الرسالة SwitchOff لإيقاف المروحة. لاحظ أننا قمنا بإعطاء القيمة 3 للمتغير currentState وذلك لأنه المروحة متوقفة في الأصل مما يعني أنها في الحالة الرابعة (تذكر أن تعداد الحالات يبدأ من 0 لأنها مصفوفة وليس من 1). فإذا قام اللاعب بتفعيل هذا المحفز سيعود إلى الحالة الأولى وبالتالي يرسل الرسالة SetSpeed1 إلى المروحة. يمكنك الاطلاع على النتيجة النهائية أيضا في المشهد scene12 في المشروع المرفق.

تمارين

1. قم بتغيير حركة إصابة الهدف في البريمج Target الموضح في السرد 15 في الصفحة 69، بحيث تصبح اندفاعا نحو الأعلى (أي في اتجاه الطلقة) وبسرعة ثابتة بدلا من الدوران وتصغير الحجم.

2. قم بتغيير بريمج تتبع الهدف TargetFollower (السرد 20 صفحة 76) بحيث يتم توجيه المقذوف أوالصاروخ نحو الهدف الأكثر بعدا عن المركبة بدلا من الهدف الأقرب.

3. قم بتغيير نظام التحكم بالمركبة ShuttleControl (السرد 21 صفحة 78) بحيث يمكن للاعب توجيه نظر المركبة مستخدما الفأرة، وذلك بقراءة موقع الفأرة وتحويل إزاحتها الأفقية إلى دوران

حول محور المركبة المحلي y.

4. قم بإضافة البريمج BulletShooter (السرد 22 صفحة 79) إلى قالب الأهداف بحيث تصبح قادرة على إطلاق الطلقات نحو الأسفل باتجاه المركبة. ستحتاج لبريمج جديد يفحص التصادم مع المركبة لا مع الأهداف، بالإضافة إلى حاجتك لقالب طلقة مختلف عن طلقة المركبة. أخيرا ستحتاج للتأكد من الجهة التي تنظر إليها الأهداف (أي الجهة التي تؤشر عليها محاور z المحلية الخاصة بها) بحيث تتأكد أنها نحو الأسفل حيث توجد المركبة، وذلك لأن الطلقة ستأخذ هذا الاتجاه. يمكنك أيضا زيادة وقت تكرار الإطلاق لجعل اللعبة أسهل أو تقليله لجعلها أصعب.

5. قم بإضافة نوع جديد من الأشياء القابلة للجمع في المثال الذي عملنا عليه في الفصل الثاني. هذا الشيء عبارة عن مضاعِف لقيمة المال Doubler وذو تأثير زمني محدود. أي أنّ اللاعب إذا قام بأخذ هذا المضاعِف ومن ثم قام بأخذ قطعة نقدية تحمل القيمة 1 فإن مقدار الزيادة في رصيد اللاعب في الحقيبة سيكون 2. يمكنك اختيار كائن اللاعب من الهرمية أثناء اللعب حتى تتمكن من رؤية التغير في قيمة money أولا بأول لتتأكد من عمل البريمج بشكل صحيح.

6. قم بإضافة محفّز جديد للضوء في الفصل الرابع بحيث يعمل على تغيير لون الضوء بين ثلاث قيم هي الأحمر والأصفر والأخضر. يمكنك الاطلاع على المحفّزات في المشهد scene12 والاستفادة منها في فهم الفكرة. ستحتاج بالطبع لإضافة بريمج أو تعديل آخر من أجل استقبال رسائل تغيير اللون بشكل صحيح.

الوحدة الرابعة: محاكاة الفيزياء

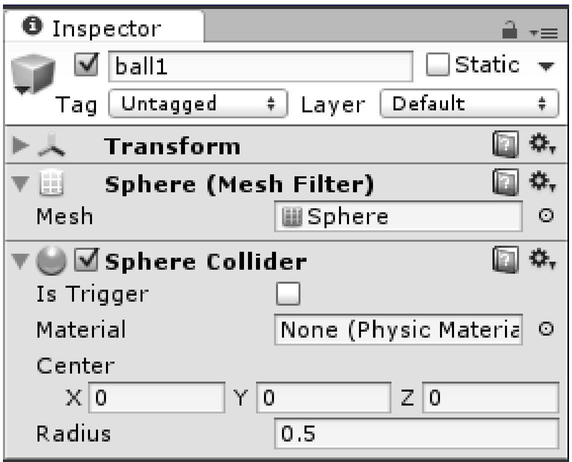
تعاملنا حتى اللحظة مع كائنات يمكن أن نعتبرها ساكنة، ذلك أنها لا تتحرك أو تستدير إلّا بفعل بريمج أو بريمجات نضيفها لها. سنتحدث في هذه الوحدة عن محاكاة الفيزياء، وهي إحدى أهم الوظائف التي يجب أن يوفرها كل محرك ألعاب. محاكاة قوانين الفيزياء الموجودة في الطبيعة كالجاذبية والتصادم تعطي كائنات اللعبة سلوكا مميزا يجعلها في النهاية أكثر متعة وواقعية.

بعد الانتهاء من هذه الوحدة يتوقع منك:

- استخدام الوظائف الأساسية للمحاكي الفيزيائي كاستخدام الجاذبية واكتشاف التصادمات
- بناء مركبات ذات خصائص فيزيائية وسلوك فيزيائي أكثر واقعية
- بناء نظام تحكم فيزيائي لشخصية اللاعب
- استخدام بث الأشعة في عملية التصويب وإطلاق النار
- عمل مقذوفات فيزيائية
- محاكاة الانفجارات والتدمير
- صناعة كائنات قابلة للكسر

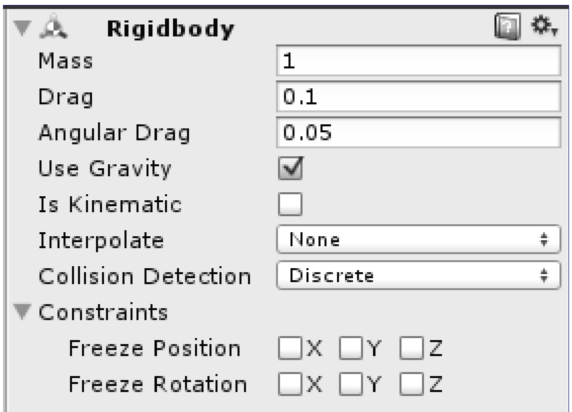
الفصل الأول: الجاذبية والتصادمات

قمنا في الوحدات السابقة بمحاكاة الجاذبية عن طريق استخدام ارتفاع مرجعي لسطح الأرض، ومن ثم تحريك العناصر بمرور الوقت نحو هذا المرجع. أما في هذه الوحدة فسنقوم باستخدام المحاكي الفيزيائي الخاص بمحرك Unity لتحقيق هذا الغرض. المكوّنان الأساسيان في عملية المحاكاة الفيزيائية في Unity هما مكوّن التصادم Collider ومكوّن الجسم الصلب Rigid Body. لعل الأول مألوف لنا كونه كان متواجدا في كافّة كائنات الأشكال الأساسية التي نضيفها للمشهد، فلو تفحّصت نافذة الخصائص عند اختيار كائن كرة ستجد مكوّن التصادم Sphere Collider كما في الشكل 43، بينما تجد Box Collider في حالة المكعب وهكذا.



الشكل 43: مكوّن التصادم Sphere Collider

من الجدير بالذكر أن الشكل المرئي للكائن منفصل عن الشكل الخاص بمكوّن التصادم، فمن الممكن مثلا أن يكون لديك كائن مكعب الشكل لكنه في نفس الوقت يتدحرج كالكرة، وكأنّه محصور داخل كرة غير مرئية. كما أنّه من الممكن تحريك مركز شكل التصادم بعيدا عن مركز الجسم المرئي، أو حتى تغيير حجم شكل التصادم لقيمة أكبر أو أصغر من الشكل المرئي. هذه التعديلات التي ذكرتها آنفا يمكن تحقيقها عن طريق تغيير قيم كل من Center و Radius الخاصة بمكوّن التصادم. على الرغم من وجود هذا المكوّن في جميع الأمثلة السابقة والمشاهد التي قمنا ببنائها، إلّا أننا لم نلحظ أي أثر له حيث كانت الكائنات تتداخل مع بعضها. السبب في ذلك يرجع إلى أننا لم نخبر Unity بأن يضع هذه الكائنات تحت تصرف المحاكي الفيزيائي، الأمر الذي يتم عن طريق إضافة مكوّن الجسم الصلب Rigid Body. بشكل عام، فإنّ اكتشاف التصادم بين كائنين لا يتم إلّا إذا احتوى كلاهما على مكوّن تصادم وأحدهما على الأقل على مكوّن جسم صلب. هذا المكوّن والموضح في الشكل 44 يجعل الكائن المُضاف إليه نشطا فيزيائيا ومتفاعلا مع غيره من الكائنات في المشهد تأثّرا وتأثيرا.



الشكل 44: مكوّن الجسم الصلب

سنقوم الآن بالتعرّف على هذا المكوّن وأهم خصائصه وكيفية الاستفادة منها تسخيرها لأهدافنا، ولتكن البداية مع مشهد كالذي في الشكل 45. جميع الكائنات في المشهد تمتلك مكوّن التصادم بشكل افتراضي، فكل ما علينا إذن هو أن نضيف مكوّن الجسم الصلب للكائنات التي نريد أن نجعلها نشطة من الناحية الفيزيائية، وهي هنا الكرات الأربع.

لإضافة مكوّن الجسم الصلب إلى كائن ما، قم باختيار الكائن من الهرمية ومن ثم اذهب للقائمة Component > Physics > Rigid Body.



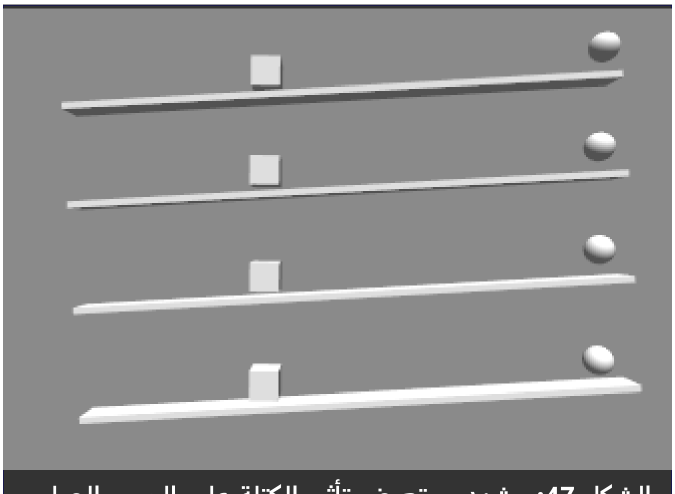
الشكل 45: مشهد بسيط لاستعراض بعض خصائص الأجسام الصلبة

أولى الخصائص التي سنتعرف عليها هي مقاومة الهواء Drag، وبناء عليها يتم حساب كمية الحركة التي يمكن للجسم أن يقوم بها بعد انتهاء تأثير كافّة القوى الخارجية عليه. بزيادة هذه القيمة تزيد المقاومة التي يتعرض لها الجسم خلال حركته، مما يجعل فقدانه للسرعة يتم في وقت أقصر وبالتالي لا يلبث أن يتوقف عند زوال المؤثر. حتى نرى تأثير هذه المقاومة، لنعطي الكرات الأربع قيما مختلفة لها ولتكن 0.1 للكرة العلوية ومن ثم 0.15 و 0.2 و 0.25 لما يليها من كرات بالترتيب. إذا شغلت اللعبة الآن ستجد أن الكرات الأربع تسقط بفعل الجاذبية على المنحدر وتنزلق أسفله قبل أن تبدأ بالحركة على المسار. هذه الحركة ستظهر بوضوح تأثير مقاومة الهواء حيث سنرى أن الكرة ذات المقاومة الأقل ستستمر في الحركة إلى أن تسقط عن حافّة المسار، بينما تتوقف الكرات الأخرى بعد قطع مسافات متفاوتة تبعا لمقاومة كل منها. نتيجة هذه الحركة تظهر في الشكل 46 كما يمكنك تجربتها في المشهد scene13 في المشروع المرفق.



الشكل 46: تأثير مقاومة الهواء على حركة الكائنات. انتبه إلى أن الكرات العلوية ذات مقاومة هواء أقل من السفلية

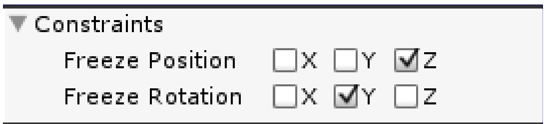
ثاني الخصائص المهمّة في مكوّن الجسم الصلب هو الكتلة Mass. كما نعرف فالكتلة تتناسب طرديا وبشكل خطي مع الوزن، وهذا الأخير هو القوة التي تجذب بها الأرض الأجسام نحو مركزها. تذكر دائما أن الوزن يمثل قوة الجذب ولا علاقة له بسرعة الجذب. لنرى ذلك على أرض الواقع (أو لنقل أرض الواقع الافتراضي) لنقم ببناء مشهد كالذي في الشكل 47، والذي يحتوي على أربعة مكعبات ذات كتلة تساوي 0.25 أي 250 جراما لكل منها، بالإضافة لأربع كرات متفاوتة في الكتلة ولتكن 10 للعلوية ومن ثم 7.5 و 5 و 1 لباقي الكرات بالترتيب نحو الأسفل.



الشكل 47: مشهد يستعرض تأثير الكتلة على الجسم الصلب

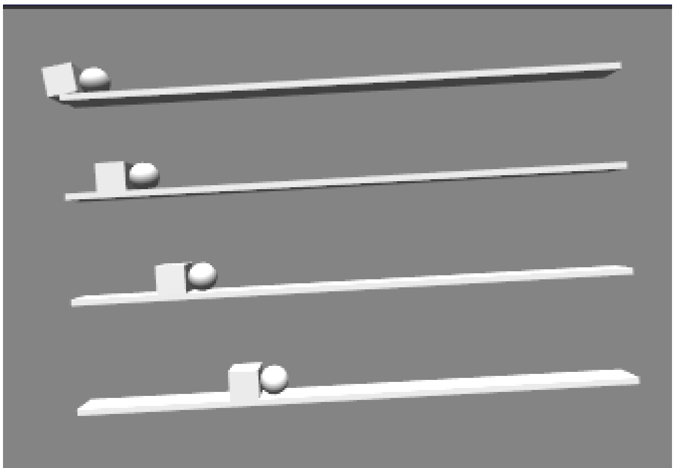
ما نتوقعه الآن عند تشغيل اللعبة هو تحرك الكرات الأربع على منحدراتها باتجاه المكعبات ومن ثم الاصطدام بهذه المكعبات وتحريكها لمسافة ما قبل التوقف تماما أو السقوط مع المكعب عن حافّة المنحدر. قبل ذلك نود القيام بتعديل بسيط وهو حصر التأثير الفيزيائي على محورين وهما x و y، فلا

نريد أن تسقط الكرات عن جانب المنحدر بعد اصطدامها، كما لا نريد أن تستدير المكعبات بفعل التصادم وذلك حتى نرى التأثير كاملا في الاتجاهات التي نريدها. من أجل ذلك سنلجأ للمحددات الحركة والدوران الموجودة في مكوّن الجسم الصلب Constraints. ما نريده تحديدا هو تجميد حركة جميع الكرات والمكعبات على المحور z وكذلك تجميد دورانها على المحور y. بعد إجراء هذه التعديلات يجب أن تبدو المحددات الخاصة بجميع الكرات والمكعبات كما في الشكل 48.



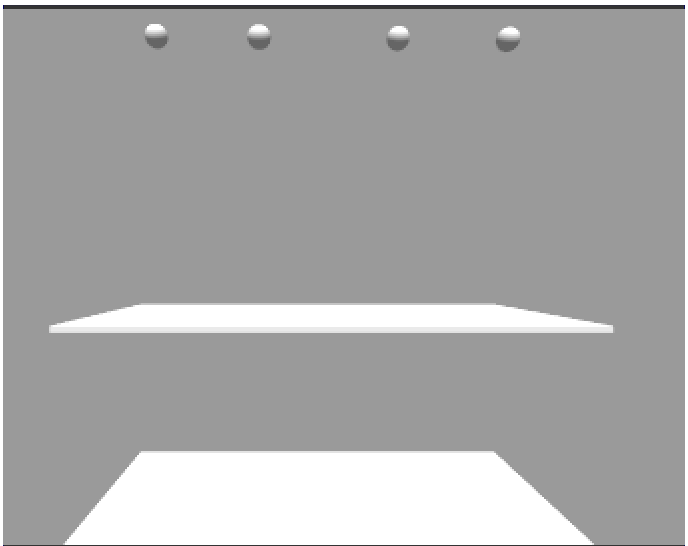
الشكل 48: محددات الحركة والدوران على الجسم الصلب

تمتلك الكرات الأربع كما ذكرنا كتلا مختلفة، لكنها تتساوى في قيمة مقاومة الهواء والتي أفترض أنك لم تغيرها حين قمت ببناء المشهد. هذا سيؤدي لأن تتحرك الكرات جميعا بسرعة متساوية على المسارات وتصطدم بالمكعبات في نفس اللحظة. بعد الاصطدام بالمكعب سيظهر فرق التأثير الذي تحدثه الكتلة، حيث أن قوة الجاذبية للكرات ذات الكتل الأعلى ستكون بطبيعة الحال أكبر، مما يجعلها تؤثر بقوة أكبر على المكعبات وبالتالي تدفعها لمسافة أبعد قبل أن تتوقف بفعل قوة الاحتكاك بين المكعب وأرضية المسار. يمكنك مشاهدة هذه النتيجة في الشكل 49 كما يمكنك تجربتها في المشهد scene14 في المشروع المرفق.


الشكل 49: تأثير كتلة الجسم الصلب، لاحظ أن الكرات ذات الكتلة الأكبر في الأعلى أزاحت المكعبات لمسافة أبعد

يقوم محاكي الفيزياء الخاص بالمحرك باكتشاف التصادمات بين الأجسام المختلفة والاستجابة لهذه التصادمات بتغيير الموقع والدوران الخاص بكائن أو أكثر مما يجعل كافّة الكائنات تتصرف كما يمكن أن

تتصرف لو كانت في الحياة الواقعية. من ناحية أخرى قد نحتاج نحن أيضا للتعامل مع هذه التصادمات حال وقوعها والاستجابة لها بطريقتنا الخاصّة التي يمكن أن نكتبها على شكل منطق برمجيّ في بريمج. فعلى سبيل المثال عندما يصطدم صاروخ بهدف ما فإننا نريد أن ندمّر كلّا من الصاروخ والهدف ونحدث انفجارا في المكان. سنتعرف في المثال التالي على كيفية عمل ذلك. سنبني مشهدا كالذي في الشكل 50 والذي يتكون من 4 كرات مختلفة الألوان سنقوم بإضافة أجسام صلبة لها، بينما اللوحان لن نصيف عليهما تلك الأجسام وسنكتفي عليهما بمكوّن التصادم.



الشكل 50: المشهد المستخدم لدراسة التعامل مع التصادمات من خلال البريمجات

قبل الانتقال للبريمجات التي سنضيفها، علينا أولا أن نقوم بتعديل بسيط على مكوّن التصادم الخاص باللوح العلوي وهو تغيير قيمة Is Trigger إلى true عن طريق اختيار المربع من نافذة الخصائص كالعادة. هذا الخيار يعني أنّ مكوّن التصادم المذكور هو عبارة عن محفّز، وبالتالي فإن محاكي الفيزياء سيكتشف التصادم بينه وبين أيّ كائن آخر ويخبرنا بوقوعه، ولكنّه من ناحيته لن يقوم بأي تغيير على الأجسام المتصادمة. بكلمات أخرى يمكن القول أنّ الكرات ستسقط وتمر من خلال اللوح العلوي دون أن يوقف حركتها وتستمر إلى أن تصل للوح السفلي حيث ستحط على سطحه. لنبدأ الآن مع بريمج يقوم بمعالجة التصادمات بين الكرة وبين الأجسام الأخرى، وهو البريمج ColorBall الموضّح في السرد 40.

```
using UnityEngine;
using System.Collections;

public class ColorBall : MonoBehaviour {

	//لون الكرة
	public Color color;

```

```
    void Start () {
        renderer.material.color = color;
    }

    void Update () {

    }

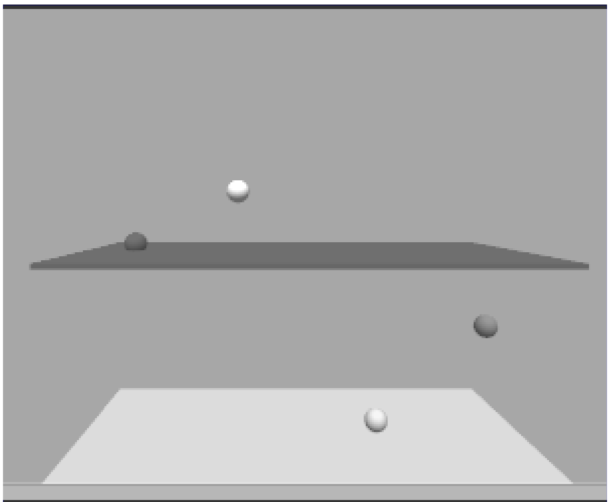
    //تقوم هذه الدّالة بمعالجة التصادمات مع الأجسام الأخرى
    void OnCollisionEnter(Collision col){
        //قم بتغيير لون الجسم الآخر الذي تم التصادم معه
        col.collider.renderer.material.color = color;
    }

    //تقوم هذه الدّالة بمعالجة التصادم مع المحفّزات
    void OnTriggerEnter(Collider col){
        col.renderer.material.color = color;
    }
}
```

السرد 40: البريمج الخاص بمعالجة التصادمات بين الكرة وغيرها من الكائنات

يسمح لنا هذا البريمج باختيار لون من نافذة الخصائص ويقوم بإسناد هذا اللون لخامة الكرة عند بداية التشغيل. لنقم مثلا بتحديد أربع ألوان مختلفة لهذه الكرات، كالأحمر والأصفر والأخضر والأزرق. لدينا دالّتان في البريمج خاصتان بمعالجة التصادمات، وكل واحدة منهما تختص بنوع مختلف من هذه التصادمات. الدّالة الأولى وهي ()OnCollisioEnter والتي يتم استدعاؤها حين تصطدم الكرة مع كائن آخر يحمل مكوّن التصادم. عند استدعاء هذه الدّالة يتم تزويدنا بمتغير من نوع Collision والذي أسميناه هنا col، هذا المتغير يحتوي على معلومات تهمنا عن التصادم الحاصل، والتي من أهمها الكائن الآخر الذي وقع التصادم معه والذي نصله من خلال col.collider. في حالة هذا البريمج كل ما نقوم به ببساطة هو تغيير لون الخامة الخاصة بالكائن الآخر بحيث تصبح مطابقة للون الكرة. الأمر نفسه ينطبق على ()OnTriggerEnter والتي يتم استدعاؤها عند التصادم مع محفّز، وهو في هذه الحالة اللوح العلوي. نظرا لأن التصادم مع المحفّز لا يترتب عليه أية آثار فيزيائية على الكائنات المتصادمة، فلا نحتاج معلومات مفصلّة عن التصادم كما هو الحال مع ()OnCollisionEnter بل يكفينا أن نعرف بحدوثه. لهذا نلاحظ أن المتغير المزوّد للدالة هنا مختلف وهو من نوع Collider، أي هو ببساطة مرجع مباشر لمكوّن التصادم في الكائن الآخر أي المحفّز. ما نقوم به هنا أيضا هو تغيير لون خامة الكائن ليطابق لون الكرة.

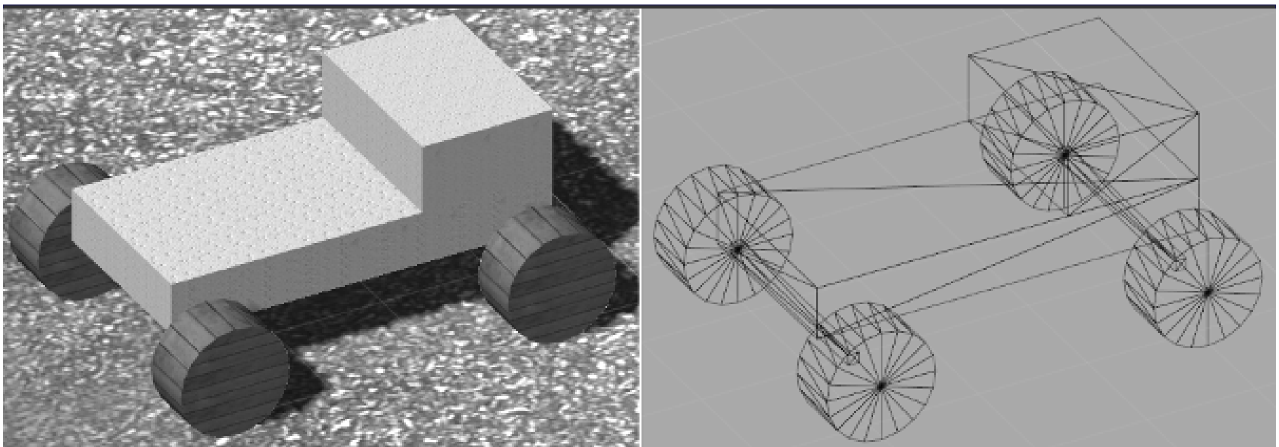
بقيت خطوة أخيرة قبل تشغيل اللعبة وهو جعل سرعات السقوط للكرات الأربع متباينة، بحيث تصطدم بالألواح في أوقات مختلفة مما يسمح لنا برؤية التأثير على لون اللوح. يمكن تغيير سرعة السقوط بتغيير قيمة مقاومة الهواء Drag على جميع الكرات، حيث كلما زدنا القيمة قلت سرعة السقوط. الشكل 51 يعرض لقطة للمثال أثناء تشغيل اللعبة، ويمكن معاينة المثال أيضا في المشهد scene15 في المشروع المرفق.



الشكل 51: معالجة التصادمات برمجيا عن بتغيير الألوان

الفصل الثاني: المركبات الفيزيائية

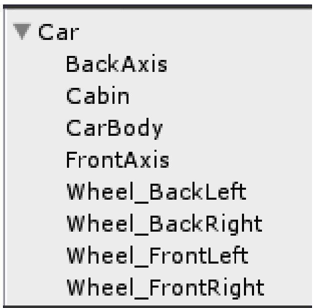
قمنا في الفصل الثاني من الوحدة الثانية على بتطبيق نظام إدخال ألعاب سباق السيارات وصنعنا سيارة بسيطة قمنا ببرمجة جميع وظائفها كالتسارع والمكابح والتوجيه بأنفسنا. أمّا في هذا الفصل فسوف نستعين بالمحاكي الفيزيائي من أجل بناء مركبة أكثر واقعية وستكون ذات أربع عجلات مع نوابضها الخاصة بامتصاص الصدمات. لنقم أولا ببناء هيكل السيارة باستخدام الأشكال الأساسية وبعض الخامات المناسبة بحيث يبدو كما في الشكل 52.



الشكل 52: سيارة بسيطة تم تركيبها باستخدام الأشكال الأساسية

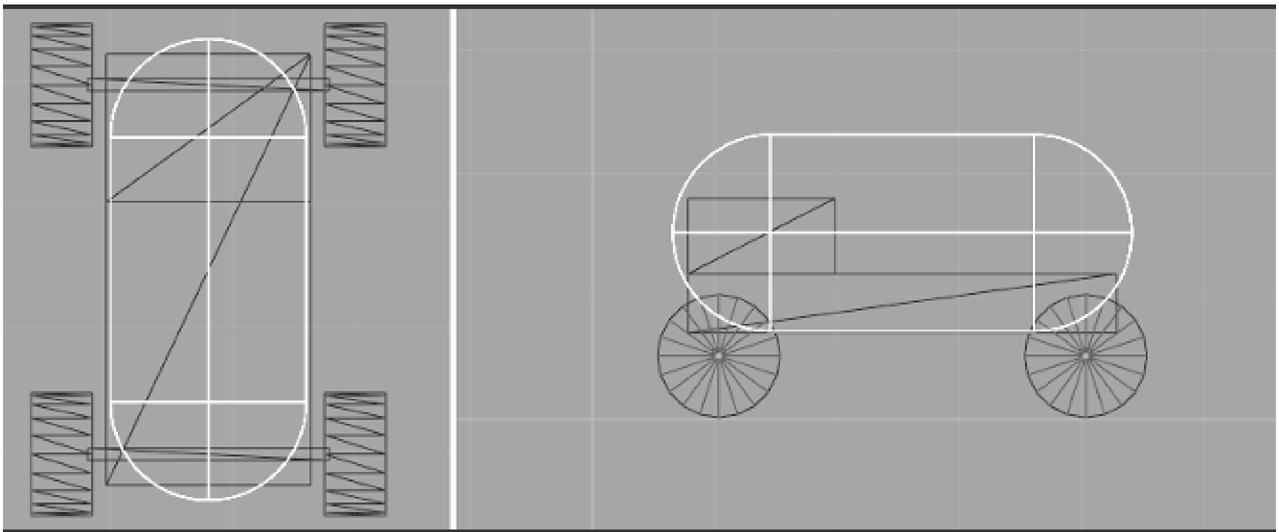
يمكن بناء المحاور الخلفية والأمامية للعجلات باستخدام مكعبات، وكذلك الأمر بالنسبة لقمرة القيادة والجسم الرئيسي. بالنسبة للعجلات يمكن استخدام أسطوانات لبنائها. بعد الفراغ من بناء المركبة علينا

أن نقوم بإزالة جميع مكوّنات التصادم Collider عن الكائنات المستخدمة في عملية البناء باستثناء المحورين الأمامي والخلفي. لتحقيق أداء أفضل، سنضيف مكوّن تصادم خاص بالمركبة بعد قليل. المهم الآن قبل الانتقال للخطوة التالية هو إضافة كائن فارغ وجمع كافّة هذه الأشكال في داخله بحيث تبدو هرمية المركبة كما في الشكل 53. من المهم عند إضافة الكائن الفارغ أن تتأكد أنه في نقطة الأصل وأن كافّة أجزاء السيارة مرتبة أيضا حل نقطة الأصل، بحيث تكون كلها مجتمعة حول مركز واحد في المنتصف.



الشكل 53: إضافة أجزاء المركبة كأبناء لكائن فارغ

سنضيف الآن مكوّن التصادم لهذا الكائن الفارغ الذي أضفناه، سنستخدم لهذا الغرض مكوّنا على شكل كبسولة Capsule Collider والذي يتميز بأنه يمنع السيارة من أن تقف على ظهرها حال انقلابها وبالتالي تستمر في الدحرجة حتى تعود لتقف على عجلاتها مجددا. لأجل ذلك يجب أن تضاف هذه الأسطوانة بحيث تمتد على طول السيارة من الأمام للخلف، وأن ترتفع قليلا عن مستوى الأرض إلى أن يلامس سطحها السفلي الجزء الأسفل من جسم السيارة. الخطوة المهمة الأخرى فيما يتعلق بالتصادم هو أن نزيد حجم مكوّنات التصادم على المحورين الأمامي والخلفي، ويجب أن تكون هذه الزيادة تحديدا على المحور x مما يجعل مكوّن التصادم الخاص بكل محور أطول من المحور نفسه: فالمحور الأصلي (المرئي) ينتهي عند الأوجه الداخلية للعجلات، بينما يجب أن يمتدّ طول مكوّن التصادم ليلامس الأسطح الخارجية للعجلات. إن كنت تتساءل عن أهمية هذه الخطوة، فهي بالتجربة ضرورية لتجنب أن تغوص العجلات عرضيا تحت سطح الأرض وبالتالي تعلق السيارة في حالة لا يمكن التحكم بها وهو ما لا نريده. بالتالي في هذه الحال إذا سقطت السيارة على جانبها فإنّ أول ما يلامس الأرض هو مكوّن التصادم الخاص بالمحور فيوقف عملية السقوط قبل أن تنزل العجلة تحت مستوى سطح الأرض. الشكل 54 يوضح الإعداد الصحيح لمكوّن التصادم الخاصة بالسيارة.



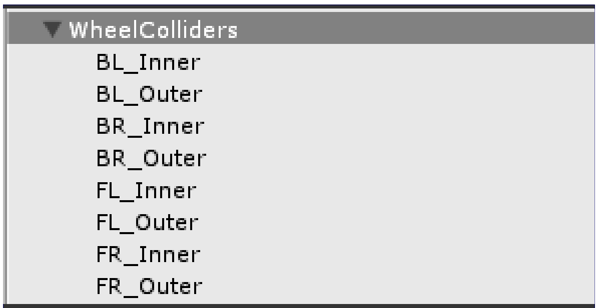
الشكل 54: إضافة مكوّن تصادم على شكل كبسولة إلى كائن السيارة

المكوّن الآخر الذي ينبغي علينا أن نضيفه للكائن الجذري لسيارتنا هو الجسم الصلب Rigid Body. سنحتاج لكتلة واقعية تناسب سيارة بهذا الحجم ولتكن مثلا 1500 كيلوجرام. سنحتاج أيضا لمقاومة هواء عالية نسبيا بحيث تعطي اللاعب شعورا بثقل وزن السيارة حين قيادتها حيث يجب أن تتوقف بعد مسافة قصيرة من توقف الضغط على دواسة الوقود. سنحتاج لهذا الأمر القيمة 0.25 لمتغير مقاومة الهواء Drag وقيمة 0.75 لمقاومة الهواء الخاصة بالدوران Angular Drag. أمر آخر مهم وهو تغيير موقع مركز الثقل الخاص بالسيارة والذي يضعه Unity افتراضيا في مركز الكائن، بيد أننا هنا نريد أن نمنع الانقلاب السهل للسيارة عند الانعطاف بالتالي سنقوم بتحريك مركز الثقل نحو الأسفل قليلا. قم بإضافة كائن فارغ آخر داخل السيارة وأسمه CenterOfMass ثم قم بتحريكه ليكون في الموقع (0.2 , 0.5- ,0) داخل جذر السيارة الرئيسي. سنقوم لاحقا باعتماد موقع هذا الكائن كمركز للثقل من خلال بريمج خاص بهذا الغرض.

سنحتاج الآن لعجلات حقيقية الأداء لسيارتنا، حيث أن العجلات تعتبر الجزء الأهم في عملية محاكاة المركبات الفيزيائية. السبب في ذلك هو أننا من خلال هذه العجلات سنكون قادرين على تطبيق عزم المحرك والمكابح والتوجيه إضافة إلى النوابض. من الأمور الخاصة بمحرك Unity والتي قد لا تنطبق بالضرورة على غيره هو عملية الفصل بين كائن العجلة المرئية وكائن محاكاة العجلة المقابل له. من أجل ذلك علينا أن نضيف أربع كائنات فارغة أخرى للسيارة ومن ثم نضيف لكل منها مكوّن محاكاة العجلة ويسمى Wheel Collider، مما يترك العجلات المرئية دون أي مكوّن تصادم.

يفضّل عند إضافة مكوّنات محاكاة العجلات أن تتم إضافتها داخل كائن فارغ آخر بداخل السيارة، بحيث يحتوي هذا الكائن فقط على أربعة أبناء يمثل كل منها عجلة من عجلات السيارة ويحمل مكوّنا واحدا فقط هو Wheel Collider. بما أن سماكة مكوّن تصادم العجلات في Unity هي صفر، فسنحتاج لإضافة اثنين من هذه المكوّنات على كل عجلة من عجلات السيارة العريضة نوعا ما، بحيث يكون أحدها على

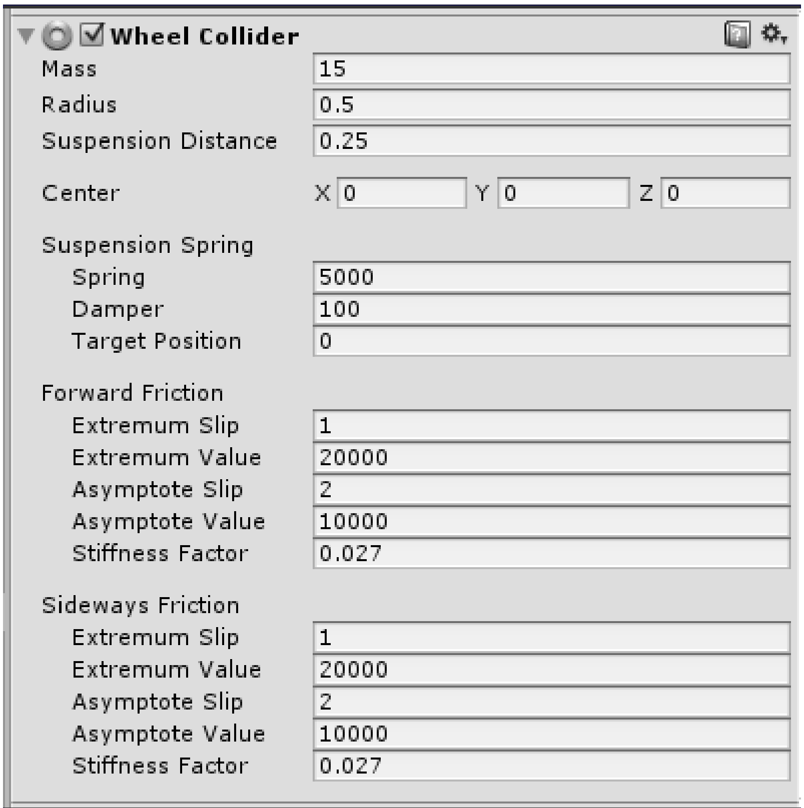
الوجه الخارجي للعجلة والآخر على الوجه الداخلي لها. معنى ذلك أننا سنضيف كائنا فارغا كابن للسيارة ولنسمه مثلا WheelColliders ومن ثم سنضيف له ثمانية كائنات فارغة أخرى كأبناء ونضيف لكل منها مكوّن تصادم العجلات Wheel Collider كما في الشكل 55.



الشكل 55: كائنات فارغة تستخدم لإضافة مكوّنات تصادم العجلات. لاحظ أن الأسماء تمثل مواقع المكوّنات بالنسبة للعجلات المرئية

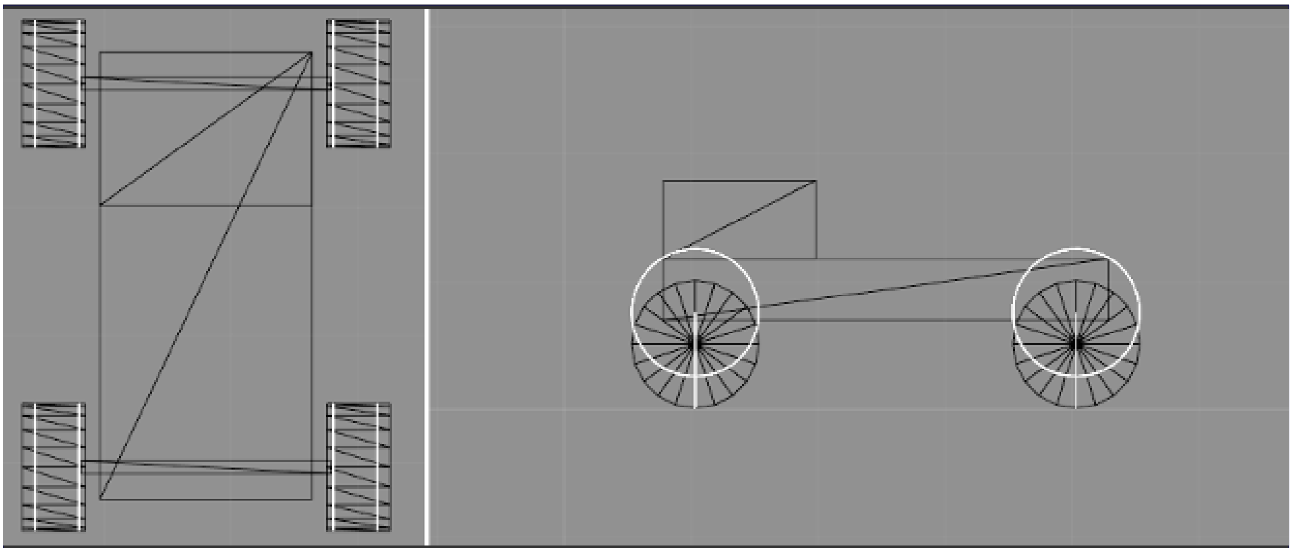
بعد إضافة مكوّن تصادم العجلات لكل واحد من هذه الكائنات الفارغة الثمانية، علينا أن نقوم بضبط خصائصها المختلفة حتى تعمل بالصورة المطلوبة، يمكنك ضبط قيم هذه الخصائص مستعينا الشكل 56.

يمكنك اختيار عدد من الكائنات دفعة واحدة ورؤية كائناتها المشتركة في نافذة الخصائص. عند ضبط القيم في هذه الحالة فإن التغيير سيطبق على كافة الكائنات التي قمت باختيارها



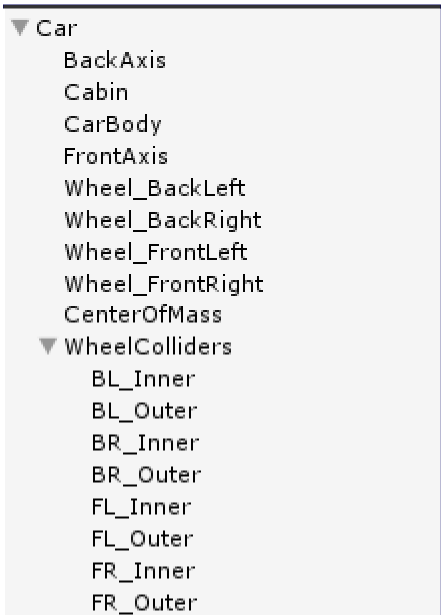
الشكل 56: ضبط قيم مكوّن تصادم العجلات

لدينا هنا مجموعة من الخصائص ذات الأهمية مثل كتلة العجلة والتي قمنا بضبطها على 15، وبما أننا نستخدم مكوّني تصادم لكل عجلة، فستكون الكتلة الإجمالية لكل عجلة من عجلات السيارة هي 30. بالنسبة لنصف القطر radius يمكننا معايرته يدويا بحيث ينطبق على العجلة المرئية للسيارة. أخيرا لدينا مسافة حركة النابض والتي ضبطناها على 0.25 أي 25 سنتيمترا في الحالة غير المضغوطة للنابض. هناك عدد من فئات المتغيرات التي تحتوي على قيم مهمة لعملية المحاكاة مثل Suspension Spring و Forward Friction والتي قد تحتاج أحيانا وقتا طويلا للتجربة حتى تجد القيم المناسبة. يمكنك الاطلاع على ملفات مساعدة Unity وبعض مصادر الإنترنت لمعرفة الطريقة الأمثل لضبط هذه القيم. بعد الانتهاء من عملية الضبط يجب وضع مكوّنات التصادم في مواقعها الصحيحة كما في الشكل 57.



الشكل 57: ضبط مكوّنات تصادم العجلات في مواقعها الصحيحة حيث تظهر المكوّنات باللون الأبيض

يمثل الخط العمودي في مكوّن تصادم العجلات مسار الانضغاط لنابض العجلات، بينما تظهر الدوائر البيضاء مكان العجل حين يكون النابض مضغوطا إلى حده الأقصى. آخذين بعين الاعتبار الحقائق السابقة، ينبغي علينا أن نضبط مواقع مكوّنات التصادم بحيث يكون الطرف السفلي من خط مسار النابض ملامسا لسطح الأرض، مما يجعل المكوّن ككل مرتفعا قليلا عن العجلة المرئية. بعد الانتهاء من تجميع كائنات السيارة ستظهر في الهرمية كما في الشكل 58.



الشكل 58: التركيبة النهائية لكائن السيارة

بهذا تصبح السيارة جاهزة لنتحكم بها، وبالتالي نحتاج لمجموعة من البريمجات وأولها PhysicsCarDriver والذي يظهر في كل من السرد 41 والسرد 42 والسرد 43. يمثل هذا البريمج الوظائف الأساسية للسيارة منفصلة عن كل ما له علاقة بمدخلات اللاعب، مما يجعله قابلا للاستخدام من قبل بريمجات أخرى كالذكاء الاصطناعي مثلا. نظرا لطول البريمج قمت بتقسيمه على 3 أجزاء حيث يظهر الجزء الأول منه في السرد 41، والذي يمثل المتغيرات التي نحتاجها.

```
using UnityEngine;
using System.Collections;

public class PhysicsCarDriver : MonoBehaviour {

	//مكوّنات تصادم العجلات الأمامية
	public WheelCollider[] frontWheels;

	//مكوّنات تصادم العجلات الخلفية
	public WheelCollider[] backWheels;

	//مركز ثقل السيارة
	public Transform centerOfMass;

	//العزم الأقصى للمحرك
	public float maxMotorTorque = 9500;

	//عزم المكابح
	public float brakesTorque = 7500;

	//أقصى زاوية استدارة لتوجيه العجلات يمينا ويسارا
	public float maxSteeringAngle = 20;

	//سرعة استدارة العجلات لليمين اليسار أثناء التوجيه
	public float steeringSpeed = 30;

	//السرعة القصوى للسيارة مقدرة بـ كم \ ساعة
	public float maxSpeed = 250;

	//السرعة القصوى للرجوع للخلف
	public float maxReverseSpeed = 20;

	//زاوية دوران توجيه العجلات الحالية
	float currensSteering = 0;

	//سرعة السيارة القصوى مقدرة بدورة في الدقيقة
	float maxRPM, maxReverseRPM;

	//متغيرات استقبال المدخلات
	bool accelerateForward,
		accelerateBackwards,
		brake, steerRight, steerLeft;

```

السرد 41: المتغيرات التي سنحتاج إليها في بريمج التحكم بالسيارة

لمر بشكل سريع على هذه المتغيرات التي نراها. لدينا في البداية مصفوفتان من نوع WheelCollider والتي سنستخدمها لتخزين كافّة العجلات الفيزيائية للسيارة. الهدف من فصل العجلات الأمامية والخلفية عن بعضها هو أننا سنقوم بعملية التوجيه باستخدام العجلات الأمامية فقط وهذا هو الحال في السيارات الحقيقية كما نعرف. المتغير الثاني وهو مركز ثقل السيارة centerOfMass والذي سنستخدمه كمرجع لكائن فارغ قمنا سابقا بإضافته بهدف إزاحة مركز ثقل السيارة نحو الأسفل لمنعها من الانقلاب عند الانعطاف. نأتي بعد ذلك على عزم كل من المحرك والمكابح maxMotorTorque و brakesTorque، والعزم فيزيائيا مقياس لحساب قدرة قوة معيّنة على تحريك جسم ما بشكل دائري، وهو تماما ما يفعله كل من المحرك والمكابح بالعجلات، مع فرق أنّ عزم المكابح يسعى لإيقاف السيارة وذلك بإلغاء عزم المحرك. المتغيران maxSteeringAngle و steeringSpeed يساعداننا في عملية التوجيه، وسنستخدمهما بطبيعة الحال مع العجلات الأمامية فقط. أخيرا وليس آخرا لدينا متغيرا السرعة القصوى نحو الأمام والخلف maxSpeed و maxReverseSpeed والذان يحددان أقصى سرعة يمكن أن تصلها السيارة بوحدة كم\ساعة.

بالإضافة للمتغيرات العامّة التي ذكرناها آنفا لدينا مجموعة من المتغيرات الخاصة بالاستخدام الداخلي للبريمج. أولها هو currentSteering والذي نستعمله لتخزين زاوية التوجيه الحالية للعجلات الأمامية، إضافة إلى maxRPM و maxReverseRPM اللذان يمثلان السرعة القصوى أماما وخلفا لكن بوحدة أخرى هي دورة\دقيقة. لماذا نحتاج لمعرفة السرعة بهذه الوحدة تحديدا؟ الجواب بسيط وهو أنّ مكوّن تصادم العجلات Wheel Collider لا يتعامل إلّا مع هذه الوحدة. أخيرا لدينا مجموعة من المتغيرات الخاصّة بحالة المدخلات، والتي نعرف من خلالها كيف يتم التحكم بالسيارة حاليا، فمثلا قيمة true للمتغير accelerateForward تعني أنّ اللاعب يضغط على دوّاسة الوقود بالتالي يجب أن تتحرك السيارة للأمام، وهذه الطريقة تنسحب على باقي متغيرات التحكم الأخرى. الجزء الثاني من البريمج موجود في السرد 42، والذي يوضّح الدّالتين Start() و FixedUpdate().

```
	void Start () {
		//احسب السرعة القصوى بوحدة دورة\دقيقة
		maxRPM =
			KmphToRPM(frontWheels[0], maxSpeed);
		maxReverseRPM =
			KmphToRPM(frontWheels[0], maxReverseSpeed);

		//قم بضبط مركز الثقل الخاص بالجسم الصلب للسيارة
		rigidbody.centerOfMass =
			centerOfMass.localPosition;
	}

	//Update() عند التعامل مع الفيزياء تستخدم الدّالة FixedUpdate() لتحديث الإطارات بدلا من
	void FixedUpdate () {
		//تحديث التسارع نحو الأمام أو الخلف حسب قيم متغيرات الإدخال
		if(accelerateForward){
			foreach(WheelCollider wheel in frontWheels){
				UpdateWheelTorque(wheel, maxMotorTorque);
```

```
                }

                foreach(WheelCollider wheel in backWheels){
                      UpdateWheelTorque(wheel, maxMotorTorque);
                }
                accelerateForward = false;
          } else if(accelerateBackwards){

                foreach(WheelCollider wheel in frontWheels){
                      UpdateWheelTorque(wheel, -maxMotorTorque);
                }

                foreach(WheelCollider wheel in backWheels){
                      UpdateWheelTorque(wheel, -maxMotorTorque);
                }
                accelerateBackwards = false;
          } else {
                foreach(WheelCollider wheel in frontWheels){
                      UpdateWheelTorque(wheel, 0);
                }

                foreach(WheelCollider wheel in backWheels){
                      UpdateWheelTorque(wheel, 0);
                }
          }

          //تحديث التوجيه نحو اليمين أو اليسار//
          if(steerRight){
                UpdateSteering(steeringSpeed * Time.deltaTime);
                steerRight = false;
          } else if(steerLeft){
                UpdateSteering(-steeringSpeed * Time.deltaTime);
                steerLeft = false;
          } else {
                UpdateSteering(0);
          }

          //تحديث حالة المكابح//
          if(brake){
                foreach(WheelCollider wheel in frontWheels){
                      wheel.brakeTorque = brakesTorque;
                }

                foreach(WheelCollider wheel in backWheels){
                      wheel.brakeTorque = brakesTorque;
                }
                brake = false;
          } else {
                foreach(WheelCollider wheel in frontWheels){
                      wheel.brakeTorque = 0;
                }

                foreach(WheelCollider wheel in backWheels){
                      wheel.brakeTorque = 0;
                }
          }
    }
```

```
119.
```

السرد 42: الدّالتان ()FixedUpdate و ()Start الخاصّتان ببريمج التحكم بالسيارة

بداية نقوم في الدّالة ()Start بتحويل السرعة من الوحدة المدخلة في المتغيرات العامّة وهي كم\ساعة إلى الوحدة التي ستتعامل بها داخليا وهي دورة\دقيقة، وهذه العملية تتم عبر دالّة قمنا بكتابتها خصيصا لهذا الغرض وهي ()KmphToRPM والتي سنناقشها بعد قليل. نقوم بعدها بتخزين القيم المحوّلة في المتغيرين maxRPM و maxReverseRPM. الخطوة المهمة الثانية التي ننفذها من خلال ()Start هي تحديد مركز ثقل الجسم الصلب التابع للسيارة بحيث يصبح في الموقع المحلي للمتغير centerOfMass والذي سنضبطه لاحقا من نافذة الخصائص ليرتبط بالكائن الفارغ الذي أضفناه ليقوم بهذه المهمة.

لننتقل الآن إلى دورة التحديث والتي نتعامل معها هذه المرة باستخدام ()FixedUpdate وذلك بخلاف ما تعوّدنا عليه من استخدام ()Update. الفرق بينهما هو أنّ ()FixedUpdate تُستَدعى على فترات زمنية ثابتة حتى تعطي أداء دقيقا لمحاكاة الفيزياء واكتشاف التصادمات، بمعنى آخر فإن قيمة Time.deltaTime هي نفسها في كل مرة تستدعيها من داخل ()FixedUpdate بخلاف ()Update التي تعطيك قيمة مختلفة في كل مرة. من داخل هذه الدّالة نقوم أولا بفحص قيمتي المدخلين accelerateForward و accelerateBackwards (الأسطر 59 إلى 86) وبناء عليها نقوم بتطبيق عزم المحرك على العجلات الأمامية والخلفية مستخدمين الدّالة ()UpdateWheelTorque. مهمة هذه الدّالة كما سنرى في تفاصيلها بعد قليل هي فحص حدود السرعة القصوى قبل تطبيق عزم المحرك من أجل ضمان ألا تتجاوز السيارة سرعتها القصوى التي حددناها، بعد أن يتم تطبيق العزم نعيد ضبط قيمة متغير الإدخال سواء كان accelerateForward أو accelerateBackwards إلى false. أمّا في حالة كانت قيم المدخلات لكلا المتغيرين أصلا false، فإنّ هذا يعني أننا سنطبق عزما بقيمة صفر على العجلات.

نقوم بعد ذلك باستخدام آلية مشابهة لتطبيق التوجيه وذلك في الأسطر 89 إلى 97، حيث نفحص قيمة كل من steerRight و steerLeft، فإذا كانت قيمتها true فإننا نقوم باستدعاء الدّالة ()UpdateSteering ممررين قيم التوجيه المناسبة لكل اتجاه، أو صفرا في حالة كانت قيمة كل من المتغيرين false. سنقوم بعد قليل أيضا بشرح تفصيلي للدّالة ()UpdateSteering وآلية عملها. أخيرا لدينا في الأسطر 101 إلى 117 عملية فحص وتطبيق المكابح، والتي نقوم بها هذه المرة مباشرة عن طريق تغيير قيمة wheel.breakTorque لجميع العجلات لتساوي عزم المكابح المخزن في المتغير breakTorque الخاص بالبريمج. لم نحتج هنا لفحص أية شروط بعكس حالة تطبيق عزم المحرك، ذلك أن المكابح لا تحتاج لأي شروط مسبقة، فهي تعمل فحسب، ولا شيء سيحدث لو ضغطناها والسيارة في حالة وقوف تام. بقي لنا الآن الجزء الأخير من بريمج التحكم بالسيارة والموضح في السرد 43، والذي يحتوي على باقي الدّوال المهمة لإتمام عملية التحكم.

```
	//قد السيارة نحو الأمام
	public void AccelerateForward(){
		accelerateForward = true;
		accelerateBackwards = false;
	}

	//قد السيارة للخلف
	public void AccelerateBackwards(){
		accelerateBackwards = true;
		accelerateForward = false;
	}

	//أدر عجلة القيادة لليمين
	public void SteerRight(){
		steerRight = true;
		steerLeft = false;
	}

	//أدر عجلة القيادة لليسار
	public void SteerLeft(){
		steerLeft = true;
		steerRight = false;
	}

	//قم بالضغط على المكابح
	public void Brake(){
		brake = true;
	}

	//تقوم  بفحص حدود السرعة وتطبيق عزم المحرك على العجلات
	void UpdateWheelTorque(WheelCollider wheel, float torque){
		wheel.motorTorque = torque;
		if(wheel.rpm > maxRPM || wheel.rpm < -maxReverseRPM){
			wheel.motorTorque = 0;
		}
	}

	//تقوم بتحديث زاوية توجيه العجلات الأمامية
	void UpdateSteering(float amount){
		if(amount != 0){
			currensSteering += amount;
		} else {
			//تم إفلات عجلة القيادة
			//في هذه الحالة يجب أن نعيدها للمنتصف بسرعة محددة
			//سنستخدم منطقة ميتة نعتبر خلالها أن العجلة عادة للزاوية صفر
			//وهي بين 3 و -3 درجات
			if(currensSteering > 3){
				currensSteering -=
						steeringSpeed * Time.deltaTime;
			} else if(currensSteering < -3){
				currensSteering +=
						steeringSpeed * Time.deltaTime;
			} else {
				currensSteering = 0;
			}
```

```
		}
		//قم بالتأكد من عدم تجاوز أقصى زاوية للتوجيه سواء لليمين أو لليسار
		if(currensSteering > maxSteeringAngle){
			currensSteering = maxSteeringAngle;
		}

		if(currensSteering < -maxSteeringAngle){
			currensSteering = -maxSteeringAngle;
		}
		//قم أخيرا بتطبيق الزاوية التي حسبناها على العجلات الأمامية فقط
		foreach(WheelCollider wheel in frontWheels){
			wheel.steerAngle = currensSteering;
		}
	}

	//تقوم بالتحويل من كم\ساعة إلى دورة\دقيقة باستخدام
	//قيمة نصف قطر العجلة المزودة
	float KmphToRPM(WheelCollider wheel, float speed){
		//حساب السرعة بوحد متر\ساعة
		float mph = speed * 1000;
		//حساب السرعة بوحدة متر\دقيقة
		float mpm = mph / 60;
		return mpm / (wheel.radius * 2 * Mathf.PI);
	}
}
```

السرد 43: الوظائف المساعدة الأخرى الخاصة ببريمج التحكم بالسيارة

عند استدعاء الدّالة ()AccelerateForward فإنّ ما تقوم به هو ضبط قيمة المتغير accelerateForward إلى true وكذلك ضبط accelerateBackwards إلى false منعا لحصول مدخلات متناقضة. نفس هذه الطريقة متّبعة أيضا من قبل ()AccelerateBackwards و ()SteerRight و () SteerLeft و ()Brake. بالنسبة للدّالة ()UpdateWheelTorque فإنّ ما تقوم به هو أخذ عجلة فيزيائية (أي متغير من نوع WheelCollider) بالإضافة لقيمة عزم لتقوم بتطبيقها على هذه العجلة، بعد تطبيق العزم تقوم بمقارنة السرعة الحالية للعجلة مع كل من القيمة الموجبة للمتغير maxRPM والقيمة السالبة للمتغير maxReverseRPM (القيمة السالبة هنا تعني أن اتجاه الدوران نحو الخلف)، فإذا تجاوزت سرع العجلة هذه الحدود تقوم بإعادة العزم إلى القيمة صفر حتى لا تسمح باستمرار الزيادة في السرعة وتبقيها ثابتة على قيمتها الحالية.

بالنسبة للدّالة ()UpdateSteering فإن دورها هو أن تأخذ قيمة رقمية وهي عبارة عن زاوية ستقوم بإضافتها لزاوية التوجيه الحالية للعجلات الأمامية currentSteering. إذا كانت هذه القيمة صفرا دل ذلك على أنّه لا يوجد أي مدخل للتوجيه بالتالي يجب إعادة عجلة القيادة للمنتصف وتوجيه العجلات نحو الأمام بسرعة محددة وهي steeringSpeed. إذا دخلت زاوية التوجيه المنطقة الميتة أثناء الإعادة وهي المنطقة بين 3 درجات و 3- درجات، فإنّنا نقوم بتغيير زاوية التوجيه لحظيا ونعيدها إلى الصفر. بعد إضافة القيمة المزودة عبر المتغير amount إلى قيمة زاوية التوجيه الحالية currentSteering علينا أن نقوم بالتأكد من أنّها تقع ضمن الحدود القصوى لزاوية التوجيه أي بين maxSteeringAngle يمينا و

maxSteeringAngle- يسارا، ففي حال تجاوزت الزاوية هذه الحدود نقوم بتعديل القيمة لتصبح مساوية للحد الأقصى المسموح به. أخيرا نقوم بضبط قيمة steerAngle لجميع العجلات الأمامية لتصبح مساوية للزاوية الجديدة التي قمنا بحسابها عبر الخطوات السابقة.

آخر الدوّال التي سنناقشها في هذا البريمج هي ()KmphToRPM والتي تأخذ متغيرين أحدهما عجلة فيزيائية لنستعمل نصف قطرها في حساباتنا والآخر هو قيمة السرعة التي نريد تحويلها، وهي بطبيعة الحال مقدرة بـوحدة كم\ساعة. نقوم في أول خطوة بتحويل السرعة من كم\ساعة إلى متر\دقية وذلك بضرب قيمة السرعة بـ 1000 ومن ثم قسمتها على 60 ونخزن الناتج في المتغير mpm. بهذا نكون قد وحدنا القيم حيث أن السرعة التي تستخدمها العجلات الفيزيائية تحسب بالدقيقة ونصف قطر العجلات محسوب بالأمتار. الخطوة التالية هي تحويل السرعة من متر\دقيقة إلى دورة\دقيقة. هذا التحويل يعتمد على محيط العجلة، حيث أنّ طول المحيط هو الذي يحدد كم مترا يمكن للعجلة أن تقطع في كل دورة كاملة. من أجل ذلك علينا أن نحسب محيط العجلة باستخدام قوانين الدائرة المعروفة لدينا، ومن ثم نقسم السرعة على هذا المحيط لتعطينا عدد الدورات في كل دقيقة، وهي بالتحديد القيمة التي نريد أن نرجعها من هذه الدّالة. بعد الانتهاء من هذا البريمج الطويل والمتعب علينا أن نضيفه لجذر كائن السيارة، وبذلك نصبح جاهزين للانتقال للبريمج التالي.

البريمج التالي الذي نحتاج إليه هو الذي يقوم بقراءة مدخلات اللاعب وتمريرها لبريمج التحكم بالسيارة. سيقوم هذا البريمج بقراءة مدخلات لوحة المفاتيح، ومن ثم استدعاء الدوّال المناسبة من بريمج PhysicsCarDriver. السرد 44 يوضح البريمج KeyboardCarController والذي سنستخدمه لقراءة مدخلات اللاعب.

```
using UnityEngine;
using System.Collections;

public class KeyboardCarController : MonoBehaviour {

	//متغير لتخزين مرجع لبريمج السيارة التي سنتحكم بها
	PhysicsCarDriver driver;

	void Start () {
		//قم بالبحث عن البريمج المضاف على نفس الكائن
		driver = GetComponent<PhysicsCarDriver>();
	}

	void Update () {
		//نستخدم السهمين الأعلى والأسفل من أجل الحركة للأمام والخلف
		if(Input.GetKey(KeyCode.UpArrow)){
			driver.AccelerateForward();
		} else if(Input.GetKey(KeyCode.DownArrow)){
			driver.AccelerateBackwards();
		}

		//نستخدم الأسهم الأيمن والأيسر للاستدارة يمينا وشمالا
```

```
            if(Input.GetKey(KeyCode.RightArrow)){
                  driver.SteerRight();
            } else if(Input.GetKey(KeyCode.LeftArrow)){
                  driver.SteerLeft();
            }

            //نستخدم مفتاح المسافة للضغط على المكابح
            if(Input.GetKey(KeyCode.Space)){
                  driver.Brake();
            }
      }
}
```

السرد 44: البريمج الخاص بقراءة مدخلات اللاعب واستخدمها للتحكم بالسيارة

هذا البريمج بسيط جدا كما ترى، فكل ما يفعله هو استدعاء الدّالة المناسبة من البريمج PhysicsCarDriver والموجود فعليا على نفس الكائن وهو هنا الكائن الجذري للسيارة. البريمج الأخير الذي سنضيفه للسيارة هو CarSpeedMeasure والذي مهمته قياس سرعة السيارة الحالية بوحدة كم\ساعة وطباعتها في نافذة المخرجات Console في Unity. هذا البريمج موضح في السرد 45.

يمكنك فتح نافذة المخرجات Console بالضغط بالفأرة على أسفل يسار الشاشة، علما بأن آخر سطر تمت طباعته في هذه الشاشة يظهر دوما بشكل تلقائي أسفل يسار الشاشة

```
using UnityEngine;
using System.Collections;

public class CarSpeedMeasure : MonoBehaviour {

      public WheelCollider wheel;

      void Start () {

      }

      void Update () {
            //قم بحساب السرعة وطباعتها في نافذة المخرجات
            print (GetCarSpeed());
      }

      //تقوم بتحويل السرعة من دورة\دقيقة إلى كم\ساعة
      //مستخدمة نصف قطر العجلة
      float GetCarSpeed(){
            //عدد الأمتار المقطوعة في الدقيقة
            float mpm = wheel.rpm * wheel.radius * 2 * Mathf.PI;
            //عدد الأمتار المقطوعة في الساعة
            float mph = mpm * 60;
            //عدد الكيلومترات المقطوعة في الساعة
            float kmph = mph / 1000;
            return kmph;
      }
```

```
}
```

السرد 45: بريمج لحساب سرعة السيارة بوحدة كم\ساعة وطباعتها في نافذة المخرجات بشكل مستمر

يمكنك ملاحظة أن هذا البريمج يحتاج على الأقل مرجعا واحدا لأحدى عجلات السيارة، وذلك ليتمكن من معرفة سرعة دورانه الحالية ونصف قطره وحساب سرعة السيارة بناء عليهما. نقوم هنا كما فعلنا سابقا بحساب محيط العجلة من أجل معرفة عدد الأمتار التي تقطعها في كل دورة. بقي الآن أن نضيف كاميرا للسيارة، ولحسن الحظ فإنّ البريمج CarCamera موجود لدينا حيث قمنا بكتابته في الوحدة الثانية وتحديدا في السرد 13 في الصفحة 60. فكل ما علينا إذن هو أن نضيف هذا البريمج للكاميرا وتحديد جذر السيارة كهدف التتبع الخاص بالكاميرا. بهذا تكون السيارة والكاميرا جاهزتان للعمل ويمكننا القيام بجولة في الأرجاء، لكن بالطبع ليس قبل أن تضيف أرضية تحت السيارة لتمشي عليها ولا بأس أيضا من بعض الأسطوانات على الأرض لتمثل مطبّات. هذه المطبّات ستساعدك في رؤية السلوك الفيزيائي للمركبة بوضوح حين تسير فوقها بسرعات مختلفة.

بقي الآن أن نضيف بعض التحسينات الجمالية ولكنها أيضا مهمّة. قمنا في الخطوات السابقة بتجهيز السيارة تماما ويمكن للاعب الآن قيادتها والتحكم بها بشكل كامل، إلّا أنه ليس قادرا بعد على رؤية العجلات وهي تدور أو رؤية حركتها عند الاستدارة، كما أنّه ليس قادرا على رؤية حركة جسم السيارة فوق النوابض ارتفاعا وانخفاضا. الملاحظ أن هذه الأمور كلها مرتبطة بعجلات السيارة، إذن كل ما علينا هو كتابة بريمج نضيفه للعجلات المرئية ويحركها بما يناسب الحالة التي عليها العجلات الفيزيائية، بحيث يتسق منظر السيارة مع أدائها. هذا البريمج هو CarWheelAnimator وموضح في السرد 46.

```
using UnityEngine;
using System.Collections;

public class CarWheelAnimator : MonoBehaviour {

	//العجلة الفيزيائية التي سنقرأ المعلومات منها
	public WheelCollider wheel;

	//الكائن المستعمل كمحور للتوجيه يمينا ويسارا
	public Transform steeringAxis;

	//يقوم هذا المتغير بتخزين آخر قيمة لزاوية التوجيه
	//وهذا ضروري لإعادة توجيه العجلة نحو الأمام
	//مرة أخرى
	float lastSteerAngle = 0;

	//لحفظ الموقع الأصلي للعجلة عند البداية
	Vector3 originalPos;

	void Start () {
		//قم بتسجيل الموقع الأصلي للعجلة عند بداية التشغيل
		originalPos = transform.localPosition;
	}
```

```

    void LateUpdate () {
        //نقوم بداية بتحويل السرعة من دورة\دقيقة إلى درجة\ثانية
        float rotationsPerSecond = wheel.rpm / 60;
        float degreesPerSecond = rotationsPerSecond * 360;

        //قم بالتدوير حول المحور المحلي y
        transform.Rotate(0, degreesPerSecond * Time.deltaTime, 0);

        //محور التوجيه موجود فقط في حالة العجلات الأمامية
        if(steeringAxis != null){
            //قم بإرجاع الدوران للأمام وذلك عن طريق
            //طرح قيمة الدوران المحفوظة من الإطار السابق
            transform.RotateAround(
                                steeringAxis.position,
                                steeringAxis.up,
                                -lastSteerAngle);
            //قم بتطبيق قيمة التوجيه الجديدة
            transform.RotateAround(
                                steeringAxis.position,
                                steeringAxis.up,
                                wheel.steerAngle);
            //قم بتحديث قيمة آخر زاوية توجيه حتى نقوم بطرحها في الإطار المقبل
            lastSteerAngle = wheel.steerAngle;
        }

        //افحص فيما إذا كانت العجلة تلمس الأرض
        WheelHit hit;

        if(wheel.GetGroundHit(out hit)){
            //العجلة تلمس الأرض
            //قم بتحريك العجلة نحو الأعلى بمقدار مسافة انضغاط النابض
            //مستخدما محور الفضاء
            float colliderCenter = hit.point.y + wheel.radius;
            Vector3 wheelPosition = transform.position;
            wheelPosition.y = colliderCenter;
            transform.position = wheelPosition;
        } else {
           //العجلة لا تلمس الأرض، علينا إذن أن نعيدها بشكل سلس إلى الموقع الأصلي
            Vector3 pos = transform.localPosition;
            pos = Vector3.Lerp(transform.localPosition,
                                        originalPos, Time.deltaTime);
            transform.localPosition = pos;
        }
    }
}
```

السرد 46: بريمج خاص بتحريك العجلات المرئية تبعا لحالة العجلات الفيزيائية

كل ما علينا فعله الآن هو إضافة هذا البريمج لكل واحدة من العجلات المرئية الأربع في السيارة، كما علينا أن نقوم بتحديد قيمة المتغير wheel من نافذة الخصائص بحيث ترتبط بأقرب عجلة فيزيائية لها. أخيرا علينا أن نحدد الكائن steeringAxis الذي سنستخدمه كمحور لدوران العجلات يمينا ويسارا، الأمر

الذي ينطبق فقط على العجلات الأمامية. من أجل ذلك سنعمل على إضافة كائنين فارغين جديدين للسيارة هما SteeringAxis_R و SteeringAxis_L، والذين سنستخدمهما كمحورين لتدوير العجلتين الأماميتين نحو اليمين أو اليسار. بطبيعة الحال فإن الموقع الصحيح لكل واحد من هذين الكائنين هو النهاية اليمنى واليسرى لمحور السيارة الأمامي وهو الكائن المسمى FrontAxis.

بالعودة للبريمج نلاحظ وجود متغيرين آخرين هما lastSteerAngle و الذي سنخزن فيه زاوية التوجيه الأخيرة التي قمنا بقراءتها من العجلة الفيزيائية، بالإضافة للموقع originalPos وهو موقع العجلة المرئية الأصلي في الفضاء المحلي للسيارة. بحفظ هذا الموقع يمكننا أن نعيد العجلة لمكانها الصحيح حين لا يكون النابض منضغطا. بما أننا نتعامل هنا مع بريمج مهمته الأساسية هي تطبيق تأثيرات مرئية لحركات معينة، فإننا سنقوم بالتحديث باستخدام الدّالة ()LateUpdate والتي سبق وتعاملنا معها في حالة احتجنا لتأخير تحديث البريمج عن بقية البريمجات الأخرى. المهام التي نقوم بتنفيذها داخل هذه الدّالة هي أولا تدوير عجلات السيارة حول محاورها أثناء سير السيارة وبنفس السرعة، وهذه العملية تتم عبر تحويل سرعة العجلة الفيزيائية من دورة\دقيقة إلى درجة\ثانية، ومن ثم استخدام القيمة الناتجة كزاوية لتدوير العجلة حول محورها المحلي y وهو الأمر الذي يتم في السطر 31. السبب في تنفيذ الدوران حول المحور y في هذه الحالة هو أنّ العجلات ما هي إلا أسطوانات تم تدويرها بمقدار 90 درجة، بالتالي أصبح المحور y هو محور الدوران المناسب لتدوير العجلات.

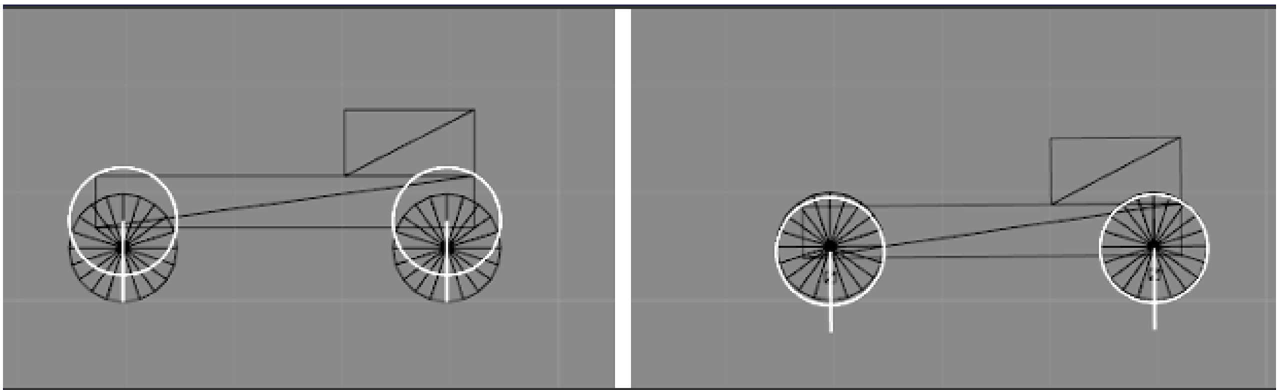
بعد ذلك نتحقق من وجود محور لزاوية التوجيه وذلك بفحص المتغير steeringAxis، فإذا لم تكن قيمته null علينا حينها أن نقوم بتدوير العجلة حول هذا المحور بناء على زاوية التوجيه التي ستعطينا إياها العجلة الفيزيائية. تتم هذه العملية عبر خطوتين، حيث نقوم في الخطوة الأولى بإعادة العجلة إلى دورانها الأصلي وذلك بتدوير العجلة حول المحور المحلي y الخاص بـ steeringAxis. قيمة الدوران التي نحتاجها هنا هي القيمة السالبة لآخر تدوير تم في الإطار السابق أي بمقدار lastSteerAngle- ، بعدها يمكننا أن نقوم باعتماد زاوية التوجيه الجديدة وذلك بتدوير العجلة حول steeringAxis بمقدار يساوي wheel.steerAngle وهي زاوية التوجيه الخاصة بالعجلة الفيزيائية. أخيرا نقوم بتخزين الزاوية الجديدة في المتغير lastSteerAngle استعدادا لطرحها في الإطار المقبل لاستقبال زاوية جديدة. جدير بالذكر أنّ قيمة المتغير steeringAxis للعجلات الخلفية هي null، مما يجعل هذه الخطوات غير منطبقة عليها.

آخر خطوة بقيت في عملية تحريك العجلة المرئية هي تحديث الموقع العمودي للعجلات اعتمادا على حالة النوابض. لو قمت بتجربة قيادة السيارة دون استخدام البريمج CarWheelAnimator فستلاحظ أن العجلات تغوص أحيانا في أرضية المشهد، وذلك يعود إلى تطبيق حركة انضغاط النوابض، حيث تقوم هذه الحركة بجعل العجلة الفيزيائية تغوص في الأرض لبعض الوقت، مما يؤدي لانخفاض العجلة المرئية كذلك. لكن هذا بطبيعة الحال ليس ما نود أن نراه، بل نريد للعجلة المرئية أن تبقى فوق سطح الأرض وأن يقوم جسم السيارة بالانخفاض نحو الأسفل ليحاكي حركة انضغاط نوابض العجلات بالشكل الصحيح.

لتحريك العجلات بالشكل الصحيح علينا أولا أن نعرف ما إذا كانت العجلة تلمس سطح الأرض أم لا، ولهذا الغرض قمنا في السطر 51 بتعريف متغير من نوع WheelHit والذي يعطينا تفاصيل عن حالة العجلة الفيزيائية. بعدها نستدعي الدّالة GetGroundHit() والتي تعطينا true في حال كانت العجلة تلمس سطح الأرض، مخزنة في الوقت ذاته تفاصيل هذا التلامس في المتغير hit والذي قمنا بتزويده مستخدمين الكلمة المفتاحية out. هذه الكلمة المفتاحية تعني ببساطة أنّ هذا المتغير تحديدا الهدف منه استخراج معلومات وقيم من الدّالة وليس تزويدها بقيم كما هي الحالة عادة في المتغيرات. أهم قيمة ستلزمنا هي hit.point والتي تمثل نقطة التماس بين العجلة وسطح الأرض. كيف سنستفيد من معرفة هذه النقطة في عملية تحريك العجلة المرئية؟ الجواب بسيط: لو قمنا بإضافة نصف قطر العجلة الفيزيائية لارتفاع هذه النقطة (أي hit.point.y) فسنحصل على ارتفاع مركز العجلة الفيزيائية عن سطح الأرض، وبالتالي نضبط ارتفاع العجلة المرئية ليصبح مساويا له كما في الأسطر 57 إلى 60. لتبسيط الأمور استخدمنا هنا إحداثيات فضاء المشهد فقط، مما ألغى أي دور لميلان السيارة عن سطح الأرض في اتجاه حركة العجلات المرئية، بيد أنّ الفرق – مع التجربة - لا يكاد يلاحظ.

قد يحدث في بعض الحالات أن "تقفز" السيارة، أي أن ترتفع عجلاتها – أو بعضها – عن سطح الأرض، مما يستوجب إعادة العجلة المرئية إلى موقعها الأصلي والذي سبق وحفظناه في المتغير originalPos. نستطيع معرفة كون العجلة مرتفعة عن سطح الأرض عن طريق الحصول على القيمة false عند استدعاء الدّالة GetGroundHit(). ما ينبغي علينا القيام به في هذه الحالة هو إرجاع العجلة لموقعها الأصلي بحركة سلسة (أي ألا تقفز فجأة للموقع الأصلي). هذه الحالة قد تتكرر كثيرا، وأعني تنفيذ خطوة تحريك أو تدوير بمقدار معين لكن عبر عدة إطارات وبمقادير قليلة في كل مرة حتى تحصل على حركة سلسة للكائنات. الدّالة الرئيسية التي لها الدور الأكبر في عملية الانتقال السلس هي Lerp() والتي يمكنك أن تجدها في عدّة أنواع من المتغيرات في Unity.

في حالة البريمج CarWheelAnimator فإننا نستدعي الدّالة Vector3.Lerp() بهدف تحريك العجلة من موقعها الحالي إلى موقعها الأصلي بشكل سلس. هذه الدّالة تحتاج لثلاث متغيرات وهي بالترتيب: موقع المصدر وهو من نوع Vector3، وموقع الوجهة وهو أيضا من نوع Vector3، ونسبة تمثل "الاستيفاء" بين الموقعين وهي رقم بين صفر و واحد من نوع float. الاستيفاء نعني به هنا هو نسبة انحياز القيمة الناتجة لقيمة الوجهة. فلو كانت هذه القيمة صفرا، فإن Lerp() ستعيد لنا نفس قيمة المصدر، ولو كانت واحد فستعيد لنا قيمة الوجهة. أمّا لو كانت نسبة الاستيفاء 0.5 مثلا، فإننا نحصل على قيمة المنتصف بين المصدر والوجهة. نستفيد من هذا السلوك في السطر 64 حيث نقوم بتغيير موقع العجلة بين موقعها الحالي transform.localPosition وموقعها الأصلي originalPos مستخدمين قيمة صغيرة جدا هي Time.deltaTime. وبما أنّ هذه القيمة صغيرة، ستكون القيمة الناتجة منحازة للمصدر بشكل أكبر، ولكنها تقترب شيئا فشيئا من الوجهة. بعد حساب هذه القيمة الناتجة نقوم بتخزينها في المتغير pos ومن ثم اعتمادها كموقع جديد للعجلة، وبذلك نكون اقتربنا ببطء من الوجهة، إلى أن نصل إليها بعد عدّة إطارات، وتحديدا بعد ثانية واحدة. الشكل 59 يمثل مواقع كل من العجلات الفيزيائية والمرئية، كما يمكنك مشاهدة النتيجة النهائية في المشهد scene16 في المشروع المرفق.



الشكل 59: نوابض العجلات في حالة الاسترخاء (يسارا) وفي حالة الانضغاط نحو الأسفل (يمينا). لاحظ أنّه في حال الانضغاط تنزل العجلات الفيزيائية تحت مستوى سطح الأرض وكذلك ينخفض مستوى جسم السيارة، بينما تبقى العجلات المرئية فوق سطح الأرض

الفصل الثالث: شخصية اللاعب الفيزيائية

سنقوم في هذا الفصل ببناء شخصية يمكن للاعب التحكم بها وتمتلك في الوقت ذاته خصائص فيزيائية. هذه الشخصية يمكن استخدامها لاحقا في بناء أنظمة تحكم لمنظور الشخص الأول أو الثالث، أو حتى أنظمة ألعاب المنصات. تتمحور الفكرة حول استعمال كائن كبسولة مضاف إليها جسم صلب، وسنقوم بالتحكم بهذه الكبسولة عن طريق إضافة القوى المناسبة كمّاً واتجاهاً. سأقوم لتوضيح الفكرة بإنشاء نظام تحكم لمنظور الشخص الأول، مما يعني أن علينا إضافة الكاميرا كابن لهذه الكبسولة في موقع الرأس (أعلى الكبسولة). بعد إضافة مكوّن الجسم الصلب Rigid Body للكبسولة، قم بضبط الكتلة (المتغير Mass) على 70 أي أن وزن الشخصية التي سنستعملها هي 70 كيلوجرام، كما علينا أن نقوم بإيقاف الدوران الفيزيائي على المحاور الثلاث، وذلك باختيار القيم الثلاث x و y و z من خانة Freeze Rotation. وهذا يعني أن المحاكي الفيزيائي والقوى المختلفة التي نطبقها على الشخصية من خلاله ستكون قادرة فقط على تحريكها ولن يكون لها أي تأثير على الدوران.

سنتبع طريقة مشابهة لبريمج التحكم بالسيارة الذي صنعناه في الفصل الثاني من هذه الوحدة، وهي جعل بريمج التحكم منفصلا تماما عن بريمج استقبال مدخلات اللاعب، ومن ثم سنقوم باستخدام بريمج الإدخال لاستدعاء الدّوال المناسبة من بريمج التحكم. البريمج PhysicsCharacter والموضح في السرد 47 يقوم عند إضافته للكبسولة بتحويلها إلى شخصية فيزيائية يمكن التحكم بحركتها.

```
using UnityEngine;
using System.Collections;

public class PhysicsCharacter : MonoBehaviour {

        //أقصى ارتفاع للقفز مقدرا بالأمتار
        public float jumpHeight = 2;
```

```

    //سرعة الحركة الأفقية
    public float movementSpeed = 8;

    //موقع أقدام اللاعب
    public Transform feet;

    //المتغيرات الخاصة بمدخلات الحركة
    bool walkForward, walkBackwards,
           strafeRight, strafeLeft, jump;

    void Start () {

    }

    public void WalkForward(){
          walkForward = true;
          walkBackwards = false;
    }

    public void WalkBackwards(){
          walkBackwards = true;
          walkForward = false;
    }

    public void StrafeRight(){
          strafeRight = true;
          strafeLeft = false;
    }

    public void StrafeLeft(){
          strafeLeft = true;
          strafeRight = false;
    }

    public void Jump(){
          jump = true;
    }

    public void Turn(float amount){
          transform.RotateAround(Vector3.up, amount);
    }

    //كما عرفنا نستخدم الدّالة FixedUpdate() عند تحديث ما يتعلق بالفيزياء
    void FixedUpdate () {
          //يمكن للاعب توجيه الجسم في اتجاه معين
          //فقط عندما يكون واقفا بقدميه على الأرض
          //أو عالقا في مكان ما بسرعة عمودية تكاد تساوي صفرا
          Vector3 velocity = rigidbody.velocity;
          if(OnGround() || Mathf.Abs(velocity.y) < 0.1f)){
                //أعد ضبط السرعة على المحورين x و z إلى صفر
                velocity.x = velocity.z = 0;

                //قم بتحديث الحركة الأفقية
                if(strafeLeft){
                      //تحرك لليسار
```

```
                        velocity += -transform.right * movementSpeed;
                        strafeLeft = false;
                  } else if(strafeRight){
                        //تحرك لليمين
                        velocity += transform.right * movementSpeed;
                        strafeRight = false;
                  }

                  if(walkForward){
                        //تحرك للأمام
                        velocity += transform.forward * movementSpeed;
                        walkForward = false;
                  } else if(walkBackwards){
                        //تحرك للخلف
                        velocity += -transform.forward * movementSpeed;
                        walkBackwards = false;
                  }
            }

            rigidbody.velocity = velocity;

            //قم بقراءة مدخل القفز
            if(jump && OnGround()){
                  //v2^2 - v1^2 = 2as نستعمل هنا معادلات المقذوفات لحساب قوة القفز
                  //v2 = 0 (عند أقصى ارتفاع)
                  //v1^2 = -2as
                  //v1 = sqrt(-2as) (sqrt تعني الجذر التربيعي)
                  float v1 =
                        Mathf.Sqrt(-2 * Physics.gravity.y * jumpHeight);

                //قم باستخدام معادلة الاندفاع p=mv مستخدما الاتجاه العلوي كاتجاه للسرعة
                  rigidbody.AddForce(
                        Vector3.up * v1 * rigidbody.mass,
                        ForceMode.Impulse);

                  jump = false;
            }

      }

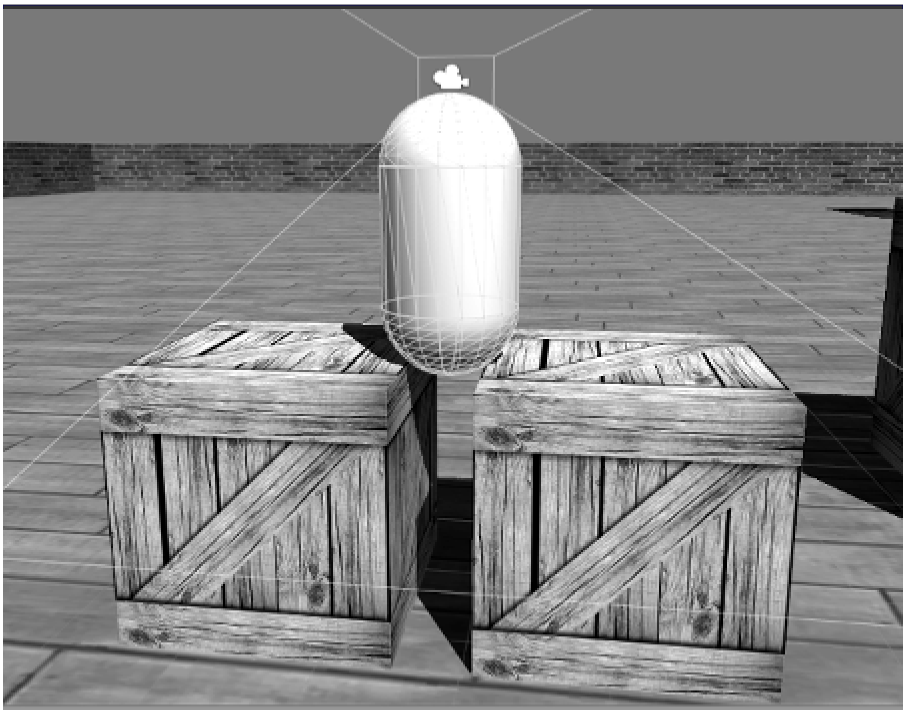
      //تقوم بحص ما إذا كانت الشخصية تقف على الأرض
      public bool OnGround(){
            //قم ببث أشعة من موقع أقدام اللاعب نحو الخلف
            //سيكون طول هذه الأشعة هو 10 سنتيمترات
            //إذا اصطدمت هذه الأشعة بالأرض أو أي كائن آخر
            //فإن ذلك يعني أن الشخصية تقف على الأرض فعلا
            if(Physics.Raycast(
                  new Ray(feet.position, -Vector3.up), 0.1f)){
                  return true;
            }
            return false;
      }
}
```

السرد 47: بريمج الشخصية الفيزيائية القابلة للتحكم

بطبيعة الحال يمكننا عند إضافة هذا البريمج لكائن الكبسولة ضبط عدد من القيم لتناسب احتياجاتنا، مثال على ذلك قيمة ارتفاع القفز jumpHeight و سرعة الحركة الأفقية (المشي) عبر المتغير movementSpeed. بالإضافة إلى ذلك علينا أن نضيف كائنا فارغا كابن للكبسولة ونضعه في أسفلها أي في موقع أقدام اللاعب ومن ثم نربطه بالمتغير feet ليمثل موقع القدمين الذي سنحتاجه في الدّالة ()OnGround في السطر 105 والتي سأبدأ بها قبل الانتقال لباقي البريمج. تخبرنا هذه الدّالة فيما إذا كانت الشخصية تقف على الأرض أم لا وذلك عن طريق استخدام بث الأشعة. بث الأشعة الذي يتم تنفيذه عن طريق الدّالة ()Physics.Raycast والتي تحتاج إلى شعاع Ray ويمكن أيضا اختياريا تزويدها بمسافة أقصى لعملية البث. تقوم هذه الدّالة ببث الأشعة في الاتجاه المحدد ومن ثم تخبرنا إذا ما اصطدمت هذه الأشعة بكائن ما. ما قمنا به هو أننا في السطر 111 قمنا بإنشاء شعاع جديد يبدأ من موقع الأقدام الذي حددناه عبر المتغير feet ويتجه في الاتجاه السالب للمتجه Vector3.up أي نحو الأسفل. قمنا أيضا بتحديد مسافة بث الأشعة بعشر سنتيمترات فقط، أي أنه بعد عشر سنتيمترات ستتوقف الأشعة سواء اصطدمت بكائن أم لا. إذا كانت القيمة التي تعيدها الدّالة ()Raycast هي true فيعني ذلك أن الأشعة اصطدمت بشيء ما، وهذا الشيء لا بد وأنه تحت أقدام اللاعب مما يعني أن اللاعب ولا شك يقف على شيء ما وبالتالي نعيد true، وما عدا ذلك نعيد القيمة false.

بالعودة لبداية البريمج، قمنا على غرار ما فعلناه في بريمج التحكم بالسيارة الفيزيائية بتحديد بعض المتغيرات الخاصة بالتحكم بحالة الشخصية. وقمنا بكتابة دالّة خاصة بكل واحد من هذه المتغيرات لتغيير قيمتها مثل ()WalkForward و ()WalkBackwards. الدّالة التي تختلف قليلا عن باقي دوال التحكم هي ()Turn والتي لا تتعامل مع متغير محدد وإنما تأخذ قيمة زاوية محددة وتدير بناء عليها كائن الكبسولة حول محوره المحلي y بنفس المقدار.

لنأتي الآن على آخر الدوال التي سنناقشها – وأكثرها أهمية بالطبع – وهي ()FixedUpdate والتي تقوم بقراءة متغيرات التحكم المختلفة وتحريك الشخصية بناء عليها. الخطوة الأولى في السطر 56 حيث نقوم بقراءة قيمة السرعة المتجهة الحالية للكائن وتخزينها في المتغير velocity. بعد ذلك نقوم بقراءة مدخلات الحركة الأفقية للاعب على المحورين x و z (أي الحركة أماما وخلفا ويمينا ويسارا). بداية فإننا نفترض أن اللاعب لن يكون قادرا على توجيه نفسه في الاتجاهات المختلفة إذا كان يطير في الهواء (أي أثناء القفز أو السقوط) ونعرف ذلك بإحدى طريقتين: الطريقة الأولى هي أن تعيد لنا الدّالة () OnGround القيمة false أي أنه لا شيء تحت أقدام اللاعب، والطريقة الأخرى هي فحص السرعة العمودية للكائن لنرى إذا ما كانت تقترب من الصفر كثيرا. الطريقة الثانية مفيدة في بعض الحالات التي قد تعلق فيها الكبسولة فوق فجوة كما في الشكل 60 مما يجعلها تبدو للدّالة ()OnGround كأنها في الهواء ولكنها في الحقيقة ليست كذلك. فإذا لم نسمح للاعب بالتحكم في الحركة الأفقية في حالة كهذه فإن الشخصية تعلق في مكانها للأبد. انتبه هنا أننا استخدمنا الدّالة ()Mathf.Abs من أجل الحصول على القيمة المطلقة للسرعة حيث تهمنا القيمة بغض النظر عن الاتجاه الموجب نحو الأعلى أو السالب نحو الأسفل.



الشكل 60: مثال على حالة لا تقف فيها الشخصية على الأرض، لكننا برغم ذلك يجب أن نكون قادرين على تحريكها أفقيا

إذا تبين لنا بعد التحقق من هذه الشروط أنه يمكن تحريك الشخصية، علينا أولا أن نقوم بتصفير سرعتها الأفقية عن طريق ضبط قيم الأعضاء x و z في متغير السرعة velocity إلى صفر. بعد أن نمر على مجموعة من الخطوات في الأسطر 62 إلى 80 نكون قد قرأنا قيم السرعة الأفقية الجديدة اعتمادا على قيم متغيرات الإدخال، ومن ثم نقوم أخيرا في السطر 83 بتخزين قيمة السرعة الجديدة في الجسم الصلب الخاص بالكبسولة. جدير بالذكر أن كل هذه الخطوات التي قمنا بها لم تمس قيمة العضو y في المتغير velocity مما يعني أن السرعة العمودية لم يحدث عليها أي تغيير حتى الآن.

الخطوة التالية هي في السطر 86، حيث نقوم بقراءة مدخل القفز جنبا إلى جنب مع عملية التحقق من كون الشخصية تقف على الأرض. إذا تحققنا من هذين الشرطين يمكننا أن نطبق القفز عن طريق إضافة قوة دافعة نحو الأعلى للشخصية ولمرة واحدة فقط، أي أشبه بضربة قوية من الأسفل تقذف الشخصية نحو الأعلى. هذه القوة يجب أن تكون بالقدر الذي يوصل الكبسولة إلى الارتفاع المحدد في المتغير jumpHeight قبل أن تبدأ بالسقوط مرة أخرى. إذن نحن نتحدث – فيزيائيا – عن مقذوف سيتم رميه للأعلى بسرعة أولية، ومن ثم تتناقص سرعته إلى الصفر عند أقصى ارتفاع للقفز. بعد ذلك يبدأ المقذوف بالسقوط نحو الأسفل بفعل الجاذبية. إذن كل ما علينا هو أن نحسب المقدار الصحيح للقوة التي سنطبقها على الكائن حتى يصل إلى الارتفاع المطلوب، وهذه العملية تتطلب الخوض في بعض التفاصيل الفيزيائية لقوانين المقذوفات والتي سأناقشها في الفقرة التالية. إذا لم تكن تحب الخوض في هذه التفاصيل يمكنك تخطي تلك الفقرة. هذه الفقرة تشرح الأسطر 91 إلى 99 من البريمج.

لحساب قوة القفز نستخدم معادلة المقذوفات $v_2^2 - v_1^2 = 2as$ ، حيث v_2 هي سرعة الجسم عند وصوله للارتفاع الأقصى، بينما v_1 هي السرعة الأولية للجسم في اللحظة التي يرتفع فيها عن سطح الأرض (سرعة الإطلاق). المتغيران الآخران في المعادلة هما a وتمثل التسارع، و s هي أقصى ارتفاع يمكن للجسم أن يصله. بتطبيق هذه المعادلة على حالة القفز التي بين أيدينا، يتبين لدينا أن المجهول الوحيد في المعادلة هو v_1 . أمّا باقي المتغيرات فهي كالتالي: عند أقصى ارتفاع للمقذوف يتوقف عن الحركة قبل أن يبدأ بالسقوط، ذلك يعني أنّ سرعته عند تلك النقطة هي صفر، بالتالي $v_2 = 0$. بالنسبة للارتفاع الأقصى للمقذوف s ، فهو نفسه أقصى ارتفاع للقفز jumpHeight وهو معروف لدينا أيضا. أخيرا فإن المقذوف يفقد من سرعته أثناء ارتفاعه نحو الأعلى بسبب تسارع وزنه نحو الأسفل، وهو بطبيعة الحال تسارع الجاذبية الأرضية وهو كما نعرف 9.8 متر\ثانية^2. بحل المعادلة لـ v_1 نحصل على التعبير $v_1 = \sqrt{-2as}$ وهو موجود في الأسطر 91 و 92. بما أنّ قوة القفز الدافعة تطبق لحظيا لمرة واحدة على شكل ضربة وتسمى فيزيائيا قوة الاندفاع ومعادلتها هي $p = mv$ حيث m هي كتلة الجسم و v هي السرعة الأولية التي حسبناها للتو. هذه المعادلة مطبقة في السطر 95 حيث نستدعي الدّالة () rigibbody.AddForce مزودين الخيار ForceMode.Impulse. بعد تطبيق القفز نعيد قيمة jump إلى false.

لنقم الآن بكتابة البريمج الذي سيقرأ مدخلات اللاعب ويقوم بناء عليها باستدعاء دوال التحكم من البريمج PhysicsCharacter. هذا البريمج هو FPSInput وهو موضح في السرد 48.

```
using UnityEngine;
using System.Collections;

public class FPSInput : MonoBehaviour {

	//سرعة النظر بالفأرة على المحورين الأفقي والعمودي
	public float horizontalMouseSpeed = 0.9f;
	public float verticalMouseSpeed = 0.5f;

	//الحد الأقصى لزاوية النظر نحو الأعلى والأسفل
	public float maxVerticalAngle = 60;

	//موقع مؤشر الفأرة في الإطار السابق
	//ويستخدم لحساب الإزاحة
	private Vector3 lastMousePosition;

	//متغير لتخزين الكاميرا
	private Transform camera;

	//بريمج الشخصية الفيزيائية الذي سنتحكم به
	PhysicsCharacter character;

	void Start () {
		character = GetComponent<PhysicsCharacter>();
		lastMousePosition = Input.mousePosition;
		//نقوم بإيجاد كائن الكاميرا في الأبناء
```

```
            camera = transform.FindChild("Main Camera");
        }

        void Update () {
            //y الخطوة الأولى هي إدارة الكبسولة حول محورها المحلي
            //بناء على الإزاحة الأفقية للفأرة
            Vector3 mouseDelta = Input.mousePosition - lastMousePosition;

            character.Turn(mouseDelta.x *
                    horizontalMouseSpeed *
                    Time.deltaTime);

            //قم بتخزين الدوران الحالي للكاميرا
            float currentRotation = camera.localRotation.eulerAngles.x;

            //قم بتحويل الدوران من المجال بين 0 و 360
            //إلى المجال بين -180 و 180
            if(currentRotation > 180){
                currentRotation = currentRotation - 360;
            }

            //قم بحسبا كمية الدوران للإطار الحالي
            float ang =
                -mouseDelta.y * verticalMouseSpeed * Time.deltaTime;

            //x الخطوة الثانية هي تدوير الكاميرا حول محورها المحلي
            //بناء على الإزاحة العمودية للفأرة
            //لكن قبل ذلك علينا فحص حدود زاوية الدوران
            if((ang < 0 && ang + currentRotation > -maxVerticalAngle) ||
                (ang > 0 && ang + currentRotation < maxVerticalAngle)){
                camera.RotateAround(camera.right, ang);
            }

            //قم بتحديث موقع الفأرة الأخير استعدادا للإطار المقبل
            lastMousePosition = Input.mousePosition;

            if(Input.GetKey(KeyCode.A)){
                character.StrafeLeft();
            } else if(Input.GetKey(KeyCode.D)){
                character.StrafeRight();
            }

            if(Input.GetKey(KeyCode.W)){
                character.WalkForward();
            } else if(Input.GetKey(KeyCode.S)){
                character.WalkBackwards();
            }

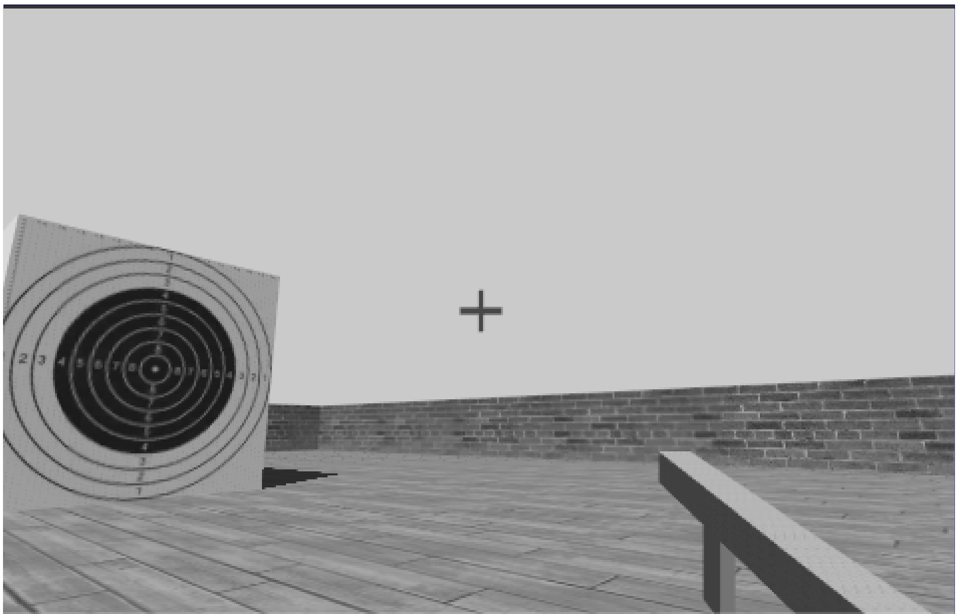
            if(Input.GetKeyDown(KeyCode.Space)){
                character.Jump();
            }
        }
}
```

السرد 48: بريمج لقراءة مدخلات اللاعب واستخدامها للتحكم بالشخصية الفيزيائية

لعلك تذكر أننا قمنا بنقاش بريمج مماثل وهو FirstPersonControl في السرد 9 (صفحة 44) حيث يتشابه البريمجان في الطريقة التي يقومان فيها بتدوير الكاميرا. الفرق بينهما هو أن FPSInput لا يملك القدرة على تحريك شخصية اللاعب مباشرة وإنما يقوم باستدعاء وظائف التحريك من البريمج PhysicsCharacter والذي يشترط وجوده على نفس الكائن، وهذا الأمر واضح في الأسطر 61 إلى 77. يمكنك الآن أن تقوم ببناء بيئة صغيرة لتجرب التحكم بهذه الشخصية الفيزيائية، ومن الأفضل أيضا أن تقوم بتعطيل مكوّن التصيير Renderer الخاص بالكبسولة. من الجدير بالذكر أنّ هذه الشخصية قادرة على دفع الكائنات الأخرى التي تحتوي على مكوّن الجسم الصلب إذا كانت ذات كتلة مناسبة، ويمكنك تجربة كل ذلك في المشهد scene17 في المشروع المرفق.

الفصل الرابع: التصويب باستخدام بث الأشعّة

سنقوم في هذا الفصل باستكمال العمل على شخصية اللاعب التي بدأناها في الفصل السابق، وذلك عن طريق إعطاء الشخصية القدرة على التصويب وإطلاق النار. وسيكون ذلك من خلال تعلّم تقنية جديدة للتصويب وهي بث الأشعّة. تذكر أننا قمنا في الفصل السابق باستخدام هذه التقنية من أجل التحقق من كون الشخصية تقف على الأرض، لكن الفرق سيكون هنا هو أننا سنحتاج معلومات إضافية عن نتيجة بث الأشعة، ولن نكتفي فقط بمعرفة حدوث اصطدام بجسم ما من عدمه. البداية ستكون ببناء بندقية بسيطة الشكل باستخدام الأشكال الأساسية كالعادة، إضافة إلى مؤشر التصويب الذي سيساعد اللاعب على تحديد هدفه. يجب أن تتم إضافة هذين الكائنين كأبناء للكاميرا بحيث يتحركان معها دائما، كما يجب أن يتم وضعهما بحيث يراهما اللاعب من الخلف كما في الشكل 61.



الشكل 61: بندقية بسيطة الشكل ومؤشر تصويب تمت إضافتهما كأبناء للكاميرا ليظهرا بمنظور الشخص الأول

كما هو متعارف عليه، سيكون ضغط اللاعب على زر الفأرة الأيسر بمثابة أمر لإطلاق النار. هذا الأمر يتم تنفيذه عن طريق بث أشعة تبدأ من موقع الكاميرا مرورا بمؤشر التصويب ثم تنطلق إلى فضاء اللعبة باعتبارها الطلقة التي خرجت من البندقية. وسنكون قادرين على معرفة ما إذا اصطدمت هذه الطلقة بشيء ما ومعالجة هذا التصادم بالطريقة المناسبة. السرد 49 يوضح البريمج RaycastShooter والذي يعتبر البريمج الأساسي للتصويب باستخدام بث الأشعّة. لكن قبل الدخول في تفاصيل هذا البريمج يجب التعرف على مجموعة من الخصائص الأساسية له، وهي الموضحة فيما يلي:

1. يقوم البريمج ببث شعاع واحد فقط في المرة الواحدة التي يتم فيها إطلاق النار.

2. هناك مسافة قصوى يمكن أن تصلها الأشعّة ويمكن تحديدها من نافذة الخصائص.

3. هناك فجوة زمنية يمكن تحديدها يجب أن تفصل بين كل شعاعين متتاليين يتم بثهما، مما يعطينا القدرة على تحديد سرعة الإطلاق لكل سلاح.

4. هناك هامش من عدم الدقة على المحورين الأفقي والعمودي لكل سلاح يستخدم هذا البريمج. هذا الهامش يمكن التعبير عنه بقياس زاوية بين الشعاع المستقيم الذي يمر من منتصف مؤشر التصويب وبين الشعاع الذي يتم بثه فعليا. قبل كل عملية بث للأشعة يتم حرف اتجاه الإطلاق أفقيا وعموديا بقيمة عشوائية بين القيمة الموجبة والسالبة للهامش، وذلك على المحورين x و y.

5. يمكن تحديد قوة تدمير لكل طلقة يتم إطلاقها عن طريق بث الأشعة، بحيث تمثل هذه القوة مدى التأثير الذي ستحدثه الطلقة على الهدف عندما تصيبه من مسافة الصفر. بزيادة المسافة بين مصدر الإطلاق والهدف سنعمل على تقليل هذا التأثير بحيث يصل إلى صفر عند أقصى مسافة تصلها الطلقة، والتي ذكرناها في النقطة رقم 2.

```
using UnityEngine;
using System.Collections;

public class RaycastShooter : MonoBehaviour {

	//أقصى مسافة يمكن للطلقة أن تصلها
	public float maxRange = 100;

	//كم من الوقت يجب أن ينقضي بين كل عمليتي إطلاق متتاليتين
	public float shootRate = 0.1f;

	//درجة الانحراف العمودية لهامش عدم الدقة
	public float verticalInaccuracy = 1;

	//درجة الانحراف الأفقية لهامش عدم الدقة
```

```
	public float horizontalInaccuracy = 1;

	//قوة الطلقة التدميرية من مسافة الصفر
	public float power = 100;

	//كائن فوهة البندقية الخاص بتحديد موقع واتجاه بث الأشعّة
	public Transform muzzle;

	//وقت آخر عملية إطلاق تمت
	float lastShootTime = 0;

	//متغير لتخزين آخر قيمتي انحراف تم استخدامهما
	Vector2 inaccuracyVector;

	void Start () {

	}

	void Update () {

	}

	//قم بمحاولة إطلاق باستخدام بث الأشعّة
	//يتم إرجاع القيمة true في حال تمت عملية الإطلاق فعليا
	public void Shoot(){
		if(Time.time - lastShootTime > shootRate){
			//قم بتوليد قيم زوايا انحراف عشوائية لهامش عدم الدقة
			inaccuracyVector.y = Random.Range(
						-horizontalInaccuracy,
						horizontalInaccuracy);

			inaccuracyVector.x = Random.Range(
						-verticalInaccuracy,
						verticalInaccuracy);

			//قم بتدوير فوهة الإطلاق مستخدما قيم الانحراف التي تم توليدها
			muzzle.Rotate(inaccuracyVector.x, 0, 0);
			muzzle.Rotate(0, inaccuracyVector.y, 0);

			//متغير خاص بتخزين بيانات نتيجة بث الأشعّة
			RaycastHit hit;

			//قم ببث الأشعّة
			if(Physics.Raycast(
					new Ray(muzzle.position, muzzle.forward),
					out hit, maxRange)){

				//قم باستبدال قيمة hit.distance
				//بقيمة التدمير التي سنعمل على حسابها

				hit.distance =
					power * (1 - (hit.distance / maxRange));
				hit.transform.SendMessage(
						"OnRaycastHit", hit,
						SendMessageOptions.DontRequireReceiver);
```

```
                }

                //قم بإرجاع دوران الفوهة إلى وضعه الأصلي عن طريق عكس قيم زوايا الانحراف
                muzzle.Rotate(-inaccuracyVector.x, 0, 0);
                muzzle.Rotate(0, -inaccuracyVector.y, 0);

                //قم بتسجيل آخر وقت تمت فيه عملية الإطلاق
                lastShootTime = Time.time;

                //قم بإبلاغ البريمجات الأخرى بحدوث عملية الإطلاق
                SendMessage("OnRaycastShoot",
                        SendMessageOptions.DontRequireReceiver);
            }
        }

        //تقوم هذه الدّالة بإرجاع قيمة آخر زاويتي انحراف لهامش عدم الدقة
        public Vector2 GetLastInaccuracyVector(){
            return inaccuracyVector;
        }
}
```

السرد 49: بريمج التصويب باستخدام بث الأشعّة

المتغيرات الأولى تمثل الخصائص التي ذكرناها قبل الخوض في البريمج وهي maxRange, shootRange, verticalInaccuracy, horizontalInaccuracy, power. أمّا المتغير muzzle فهو يمثل موقع واتجاه الأشعّة التي سيتم بثها. من أجل الحفاظ على وتيرة إطلاق النار وتنفيذ الفجوة الزمنية بين العمليات المتتالية، سنحتاج لتخزين وقت آخر عملية إطلاق وهي الغاية من تعريف المتغير lastShootTime. في كل مرة تتم فيها عملية الإطلاق نقوم بتوليد قيم انحراف عشوائية من أجل تطبيق هامش عدم الدقة، ومن ثم نقوم بتخزين هذه القيم في المتغير inaccuracyVector لنقوم باستخدامها لاحقا.

تمثّل الدّالة ()Shoot الوظيفة الأساسية لهذا البريمج، وهي تنفيذ عملية بث الأشعّة وكل ما يتصل بها. الخطوة المبدئية قبل الشروع في تنفيذ بث الأشعّة هي التحقق من انقضاء الفجوة الزمنية بين آخر عملية إطلاق نار وبين الوقت الحالي، فإذا تحقق هذا الشرط تقوم الدّالة بتوليد قيمتي زاويتي انحراف عشوائيتين بحيث تكون الأولى للانحراف الأفقي وتقع بين horizontalInaccuracy و -horizontalInaccuracy، بينما تقع الثانية بين verticalInaccuracy و -verticalInaccuracy وتختص بالانحراف العمودي. بعد توليد هذه القيم يتم تخزينها في العضوين x و y من المتغير inaccuracyVector. قبل تنفيذ الانحراف، من المفترض أن يكون الاتجاه الموجب للمحور z الخاص بالكائن muzzle مؤشرا في نفس اتجاه إشارة التصويب أي بخط مستقيم نحو الهدف. عندها نقوم بتدوير هذا الكائن على محوريه المحليين x و y بزوايا تساوي قيم الانحراف العشوائية التي قمنا بتوليدها، مما يجعل الاتجاه الأمامي للكائن ينحرف بشكل طفيف عن الهدف.

بعد تنفيذ الانحراف المطلوب أصبحنا جاهزين لتنفيذ عملية بث الأشعة، حيث نقوم في السطر 56 بتعريف المتغير hit من نوع RaycastHit والذي سنقوم باستخدامه لتخزين بيانات عملية بث الأشعة بعد تنفيذها. نقوم في الأسطر 59 إلى 61 باستدعاء ()Physics.Raycast لتنفيذ عملية الإطلاق، بحيث تنطلق الأشعة ابتداء من موقع الكائن muzzle متجهة في الاتجاه الموجب لمحوره المحلي z. لاحظ هذه المرة أننا قمنا بتزويد الدّالة ()Physics.Raycast بالمتغير hit مستخدمين الكلمة المفتاحية out، والتي تعني باختصار أنّ هذا المتغير لا يحمل أي قيمة ليتم إدخالها على الدّالة، بل إننا نتوقع من الدّالة نفسها أن تقوم بتزويدنا بقيمة معينة تقوم بتخزينها في هذا المتغير. آخر متغير نحتاج لتزويده هو أقصى مسافة يجب أن تقطعها الأشعة قبل أن تتوقف وهي maxRange في حالتنا هذه. إذا اصطدمت الأشعّة بكائن ما قبل أن تصل إلى المسافة الأقصى لها فإنه يتم تنفيذ الأسطر 66 إلى 70، حيث يعطينا المتغير hit.distance مقدار المسافة بين موقع انطلاق الأشعة وبين الهدف الذي تمت إصابته، والذي نقوم باستخدامه لحساب مقدار التدمير الذي ستحدثه الطلقة للهدف بناء على قوة التدمير من مسافة الصفر وهي معروفة لنا عبر المتغير power.

لاحظ أننا قمنا في السطر 67 بإنشاء تناسب خطي عكسي بين مقدار قوة التدمير ومقدار المسافة بين موقع إطلاق الشعاع والهدف، بحيث تصل هذه القوة إلى الصفر عند أقصى مسافة للشعاع. بعد أن نقوم بحساب قيمة التدمير فإنّ المسافة المخزنة في المتغير hit.distance لم تعد تلزمنا، لذا نقوم باستغلال هذا المتغير لتخزين قيمة التدمير التي قمنا بحسابها والتي هي أكثر أهمية بالنسبة لنا. أخيرا ينبغي أن نقوم بإعلام الهدف أنه تمت إصابته، وذلك عن طريق إرسال رسالة له تحمل الاسم OnRaycastHit ونقوم بإلحاق المتغير hit بما يحويه من بيانات إلى هذه الرسالة.

بعد اكتمال عملية الإطلاق يجب أن نقوم بإرجاع الفوهة إلى وضعها الصحيح قبل تنفيذ الانحراف، وذلك عن طريق تدويرها باستخدام القيم السالبة لقيم الانحراف الأصلية. بعد ذلك يجب أن نقوم بتسجيل الوقت الذي تمت فيه عملية الإطلاق باستخدام المتغير lastShootTime ومن ثم نقوم أخيرا بإرسال الرسالة OnRaycastShoot لإعلام البريمجات الأخرى باكتمال عملية الإطلاق. هذه الرسالة قد تنفع لتنفيذ مهام أخرى متعلقة بعملية الإطلاق مثل تشغيل صوت معين أو تشغيل تحريك ما. يمكننا أيضا الوصول لآخر قيمتين تم استخدامهما كزوايا للانحراف وذلك عن طريق الدّالة ()GetLastInaccuracyVector والتي قد تكون مفيدة في تنفيذ اهتزاز للكاميرا أو تحريك البندقية كما سنرى في البريمج RaycastShooterAnimator الموضح في السرد 50. وظيفة هذا البريمج هي القيام بتحريك كائن البندقية أثناء عملية الإطلاق وذلك باستخدام زوايا الانحراف، مما يعطي اللاعب علامة واضحة يعرف من خلالها أن الإطلاق تم بالفعل.

```
using UnityEngine;
using System.Collections;

public class RaycastShooterAnimator : MonoBehaviour {

        //مسافة الرجوع للخلف عند الإطلاق والتي تحاكي ردة فعل إطلاق الرصاصة
```

```
    public float zDistance = 0.15f;

    //متغير للوصول إلى بريمج الإطلاق الموجود على نفس الكائن
    RaycastShooter shooter;

    //الموقع الأصلي للبندقية قبل التحريك
    Vector3 originalPosition;

    //الدوران الأصلي للبندقية قبل التحريك
    Quaternion originalRotation;

    void Start () {
        shooter = GetComponent<RaycastShooter>();
        originalPosition = transform.localPosition;
        originalRotation = transform.localRotation;
    }

    void LateUpdate () {
        //قم بإرجاع البندقية ببطء وانسيابية نحو موقعها ودورانها الأصليين
        transform.localPosition =
            Vector3.Lerp(transform.localPosition,
                originalPosition, Time.deltaTime * 10);

        transform.localRotation =
            Quaternion.Lerp(transform.localRotation,
                originalRotation, Time.deltaTime * 10);
    }

    void OnRaycastShoot(){
        //تمت عملية الإطلاق، لذا نقوم بالتحريك بناء على قيمها
        Vector2 rotation =
            shooter.GetLastInaccuracyVector();
        transform.Rotate(rotation.x, 0, 0);
        transform.Rotate(0, rotation.y, 0);
        transform.Translate(0, 0, -zDistance);
    }
}
```

السرد 50: البريمج الخاص بتحريك البندقية اعتمادا على عملية الإطلاق باستخدام بث الأشعّة

الخطوة الأولى كما تلاحظ هي تخزين الموقع والدوران الأصليين للكائن وذلك حتى نكون قادرين على إعادته لمكانه بعد انتهاء عملية التحريك. المتغير المهم الآخر هو zDistance والذي يحدد مقدار الحركة على المحور z نحو الخلف عند الإطلاق. عندما يقوم اللاعب بعملية الإطلاق فإن البندقية ترتد نحو الخلف بسرعة كبيرة، وهي التي ننفذها فعليا بشكل لحظي عن طريق وضع البندقية في موقع يبعد عن موقعها الأصلي بمقدار zDistance نحو الخلف. إضافة لإرجاعها نحو الخلف، فإننا نقوم بتدوير البندقية نحو أفقيا وعموديا حول محاورها المحلية بمقدار يساوي زوايا الانحراف العشوائية التي قرأناها من البريمج RaycastShooter عن طريق الدّالة ()GetLastInaccuracyVector. بعد نقوم عبر الدّالة ()LateUpdate بإرجاع البندقية لموقعها ودورانها الأصلي بشكل سلس عن طريق استدعاء كل من ()Vector3.Lerp و ()Quaternion.Lerp. لتسريع عملية رجوع البندقية لوضعها الأصلي والتي نحتاجها في حالة إطلاق النار المتتابع، نقوم بضرب قيمة الوقت المنقضي بـ 10.

أصبح لدينا الآن شخصية لاعب فيزيائية يمكننا التحكم بها بالإضافة لامتلاك هذه الشخصية لبندقية تطلق النار عن طريق بث الأشعّة. ما تبقى علينا عمله الآن هو أن نعطي اللاعب القدرة على إطلاق النار مستخدما الفأرة وتحديدا الزر الأيسر، وهي في الواقع مهمة بسيطة ننفذها عبر البريمج GunInput والذي يقوم ببساطة بقراءة مدخل زر الفأرة الأيسر ويستدعي الدّالة ()Shoot من البريمج RaycastShooter بناء على ذلك. هذا البريمج موضح في السرد 51.

```
using UnityEngine;
using System.Collections;

public class GunInput : MonoBehaviour {

	//بضبط هذا المتغير على true فإننا لا نشترط
	//على اللاعب أن يفلت الزر بعد كل عملية إطلاق
	public bool continuous = true;

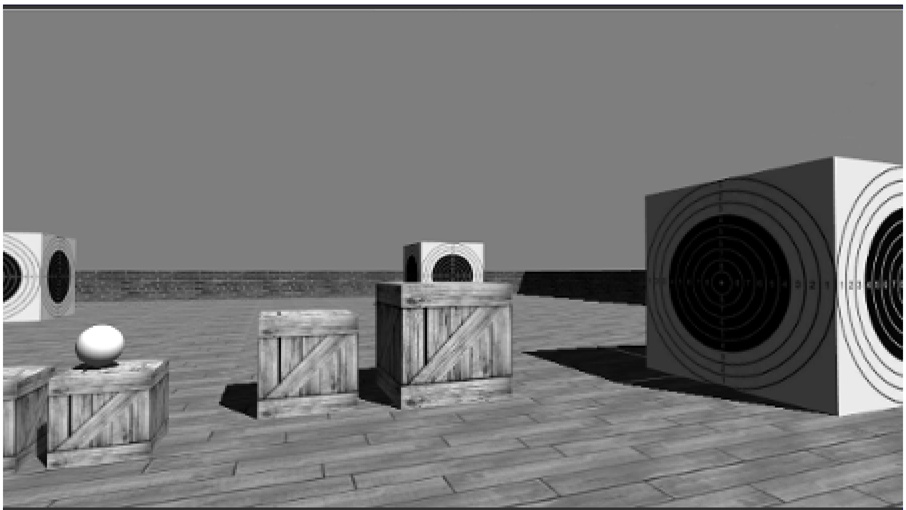
	void Start () {

	}

	void Update () {
		//قم بإرسال الرسالة Shoot بناء على مدخل زر الفأرة الأيسر
		if(continuous){
			if(Input.GetMouseButton(0)){
				SendMessage("Shoot");
			}
		} else {
			if(Input.GetMouseButtonDown(0)){
				SendMessage("Shoot");
			}
		}
	}
}
```

السرد 51: بريمج إطلاق النار عن طريق قراءة زر الفأرة الأيسر

يمكننا باستخدام المتغير continuous أن نتحكم بنوع إطلاق النار، بأن نسمح للاعب بالضغط المستمر أو نجبره على إفلات زر الفأرة قبل الإطلاق مرة أخرى. من المثير للاهتمام في هذا البريمج هو استقلاليته الكاملة عن البريمج RaycastShooter مما يجعله قابلا لإعادة الاستخدام مع أنواع أخرى من البريمجات التي تمثل الأسلحة، والتي بدورها لا تحتاج إلّا إلى القدرة على استقبال الرسالة ()Shoot. بعد الانتهاء من كتابة البريمجات الثلاثة RaycastShooter و RaycstShooterAnimator و GunInput يجب أن تتم إضافتها إلى كائن الكبسولة الذي يتحكم به اللاعب. بعد إضافة البريمجات يجب أن يتم تعيين كائن ما كفوهة للإطلاق في المتغير muzzle داخل RaycastShooter وفي حالتنا هذه يمكن أن يكون هذا الكائن هو مؤشر التصويب. أخيرا يمكنك بناء مشهد كالذي في الشكل 62 وإضافة بعض الكائنات إليه وذلك حتى تتمكن من اختبار عملية التصويب.



الشكل 62: مشهد خاص باختبار التصويب باستخدام بث الأشعّة

الخطوة التالية هي تحديد ما الذي سيحصل حين تتم إصابة الهدف. ردة فعل الهدف على الإصابة قد تختلف من هدف لآخر، فمثلا إصابة جدار ليست كإصابة لوح خشبي أو زجاجي، وهذه بدورها ليست كإصابة لاعب آخر أو حتى برميل متفجرات. هذا التنوع يترجم عن طريق الاستجابات المختلفة للرسالة OnRaycastHit والتي يتم إرسالها لكل هدف تتم إصابته حتى يستجيب لها بالطريقة المناسبة. سوف أستعرض هنا مثالين مختلفين على طريقة الاستجابة للإصابة وسأبدأ بالبريمج BulletHoleMaker الموضح في السرد 52. مهمة هذا البريمج هي رسم فتحة صغيرة (ثقب) في المكان الذي أصابته الطلقة على سطح الجسم، هذا الثقب في الحقيقة هو كائن سنستعمل له قالبا خاصا نقوم بنسخه في المكان المناسب مع كل عملية إصابة. هذا القالب يتكون ببساطة من كائن مربع quad مضاف إليه إكساء على شكل ثقب.

```
using UnityEngine;
using System.Collections;

public class BulletHoleMaker : MonoBehaviour {

        //قالب كائن الثقب الذي سنقوم باستخدامه
        public GameObject holePrefab;

        //عدد الثواني التي يجب انتظارها قبل تدمير كائن الثقب
        public float holeLife = 15;

        void Start () {

        }

        void Update () {

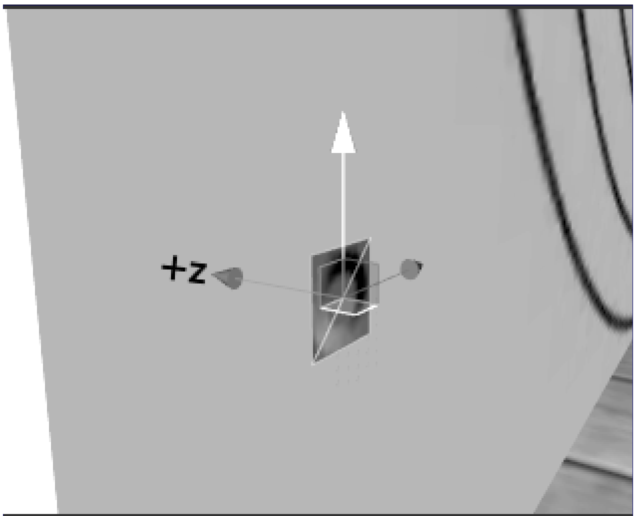
        }

```

```
20.      //قم باستقبال الرسالة OnRaycastHit ومن ثم إنشاء ثقب في مكان الإصابة
21.      void OnRaycastHit(RaycastHit hit){
22.            GameObject hole = (GameObject) Instantiate(holePrefab);
23.            hole.transform.position = hit.point;
24.            //إذا كان هناك مكوّن جسم فيزيائي صلب يجب أن نضيف الثقب كابن للهدف
25.            //مما يجعل الثقب يتحرك ويستدير مع الهدف إذا ما تحرك
26.            if(hit.rigidbody != null){
27.                  hole.transform.parent = hit.transform;
28.            }
29.            //بما أننا نستخدم كائنا من نوع quad فإننا نحتاج لأن نديره 180 درجة
30.            //نحو الداخل ليظهر سطحه المرئي، هذه حالة خاصة قد لا تنطبق بالضرورة على كائنات أخرى
31.            hole.transform.LookAt(hit.point - hit.normal);
32.            //قم بإزاحة كائن الثقب بمقدار ضئيل نحو الخارج بحيث
33.            //نضمن أن يكون ظاهرا فوق السطح المصاب
34.            hole.transform.Translate(0, 0, -0.0125f);
35.
36.            //قم باستدعاء التدمير بعد فترة محددة
37.            Destroy(hole, holeLife);
38.
39.            //قم بطباعة بعض المعلومات
40.            print (name + " took damage of " + hit.distance);
41.      }
42. }
```

السرد 52: البريمج الخاص بإحداث ثقب في مكان إصابة الطلقة

عند إضافة هذا البريمج لكائن ما فإنه يقوم بالاستجابة للرسالة OnRaycastHit عن طريق عمل ثقب في مكان الإصابة. الخطوة الأولى في الدّالة ()OnRaycastHit هي أن تقوم بعمل نسخة من القالب holePrefab ومن ثم وضعه في مكان الإصابة hit.point والذي يمثل موقع اصطدام الشعاع بالكائن في فضاء المشهد. إذا كان الكائن نشطا فيزيائيا (أي أنّه يحمل مكوّن الجسم الصلب rigid body) فإنه ينبغي علينا إضافة كائن الثقب كابن للهدف لأنه من الممكن أن يتحرك أو يستدير، ونريد أن نضمن أن يبقى الثقب في مكانه بالنسبة للهدف. السؤال الذي يطرح نفسه الآن، ما هي الوجهة الصحيحة لإدارة الثقب حتى يظهر بالشكل المطلوب؟ الإجابة المباشرة هي نحو الخارج بالنسبة للسطح الذي تمت إصابته. يسهل علينا المتغير hit العملية عن طريق توفير المتجه hit.normal، وهو متجه بطول وحدة واحدة يمتد من موقع الإصابة على سطح الهدف نحو الخارج في اتجاه عمودي على السطح. بإضافة هذا المتجه إلى موقع الإصابة فإن المتجه الناتج سيدلنا على اتجاه الدوران الصحيح لكائن الثقب.
الاختلاف البسيط هنا هو أننا نستخدم كائنا من نوع quad وهو ذو وجه مرئي واحد فقط، وهذا الوجه هو الذي تراه عندما تنظر من الاتجاه السالب للمحور z. بكلمات أخرى، حتى نظهر هذا الوجه بالطريقة الصحيحة للاعب يجب أن ندير المتجه الأمامي للكائن نحو الداخل في اتجاه متعامد على سطح الهدف المصاب كما في الشكل 63، وبالتالي فبدلا من أن نجمع المتجهين hit.point و hit.normal فإننا نقوم بطرحهما كما في السطر 31.



الشكل 63: كائن ثقب تم بناؤه باستخدام الشكل quad، بالتالي يجب أن تتم إدارته نحو الداخل ليظهر الوجه المرئي نحو الخارج

بعد إدارة الكائن في الاتجاه الصحيح، علينا التأكد من أنه سيظهر فوق سطح الهدف دائما. من أجل هذا علينا أن نحرك كائن الثقب نحو الخارج قليلا لا أن نجعله ملاصقا تماما لسطح الهدف المصاب. مسافة التحريك يجب أن تكون ضئيلة جدا بحيث لا يمكن للاعب بأي حال من الأحوال ملاحظة وجود فراغ بين الهدف والثقب الذي يفترض أنه على سطحه. في حالتنا يمكن أن نزيح الثقب نحو الخلف قليلا بمقدار 0.0125. إضافة لذلك علينا أن نبقي عدد كائنات الثقوب محدودا حتى لا يؤثر ذلك على الأداء. من أجل هذا قمنا بتحديد عمر افتراضي لكل كان ثقب تتم إضافته بحيث يتم تدميره تلقائيا بعد انقضاء مدة زمنية محددة. أخيرا تذكر أننا قمنا باستغلال العضو hit.distance من أجل تخزين قيمة قوة التدمير التي قمنا بحسابها بناء على بعد الهدف عن مركز الإطلاق. ما نقوم به في السطر 40 هو عملية طباعة لاسم الهدف ومقدار القوة التدميرية التي تلقاها من الطلقة التي أصابته، وبملاحظة هذه الأرقام ستتكون لديك فكرة عن تأثير المسافة على قوة إصابة الطلقة للهدف.

البريمج الثاني الذي سنقوم بكتابته كمستجيب للرسالة OnRaycastHit هو البريمج BulletForceReceiver والذي يختص بالكائنات التي تحتوي على جسم فيزيائي صلب. هذا البريمج موضح في السرد 53.

```
using UnityEngine;
using System.Collections;

public class BulletForceReceiver : MonoBehaviour {

	void Start () {

	}
```

```

	void Update () {

	}

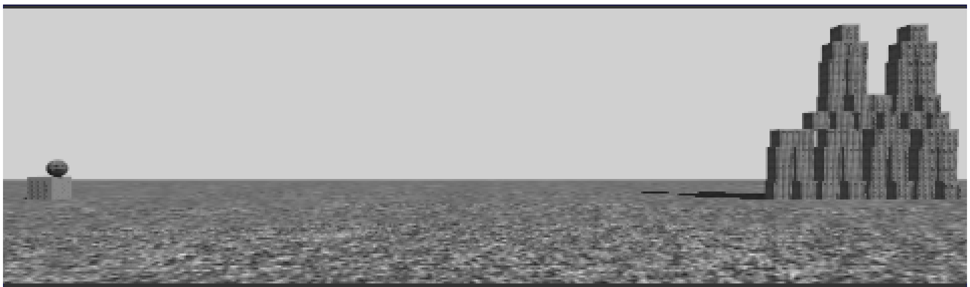
	//قم باستقبال الرسالة OnRaycastHit وترجمها إلى قوة اندفاع فيزيائية لتحريك الهدف
	void OnRaycastHit(RaycastHit hit){
		rigidbody.AddForceAtPosition(
		   -hit.normal * hit.distance, hit.point, ForceMode.Impulse);
	}
}
```

السرد 53: بريمج خاص باستقبال إصابة الأشعة وتحويلها لقوة اندفاع فيزيائية تحرك كائن الهدف

عندما يتم استقبال الرسالة OnRaycastHit من قبل هذا البريمج فإنّه يقوم بتطبيق قوة اندفاع حركة فيزيائية على الجسم الصلب الخاص بالكائن، وذلك باستخدام الدّالة ()AddForceAtPoint. ما يميز هذه الدّالة هو إمكانية تحديد نقطة معينة على الكائن لتطبيق القوة عليها، وليس بالضرورة على مركز ثقل الكائن. فمثلا عندما تكون الإصابة لصندوق ما في الجهة العلوية من أحد جوانبه، يجب أن تتركز معظم قوة الإصابة في تلك النقطة، مما يجعل الصندوق يستدير قليلا من قوة الإصابة وربما ينقلب على جانبه. الموقع الذي سنختاره لتطبيق القوة عليه هو بطبيعة الحال موقع الإصابة hit.point ومقدار القوة هو hit.distance حيث قمنا بتخزين مقدار التدمير الذي حسبناه تبعا للمسافة بين الهدف ومركز الإطلاق. أخيرا بقي علينا تحديد اتجاه القوة التي سنطبقها وهو هنا hit.normal- حيث أنّ hit.normal متعامدة على سطح الهدف نحو الخارج، بينما نحتاج لتطبيق قوة نحو الداخل لتحريك الجسم بعيدا عن اتجاه مصدر الطلقة. يمكنك مشاهدة النتيجة النهائية في المشهد scene17 في المشروع المرفق.

الفصل الخامس: المقذوفات الفيزيائية

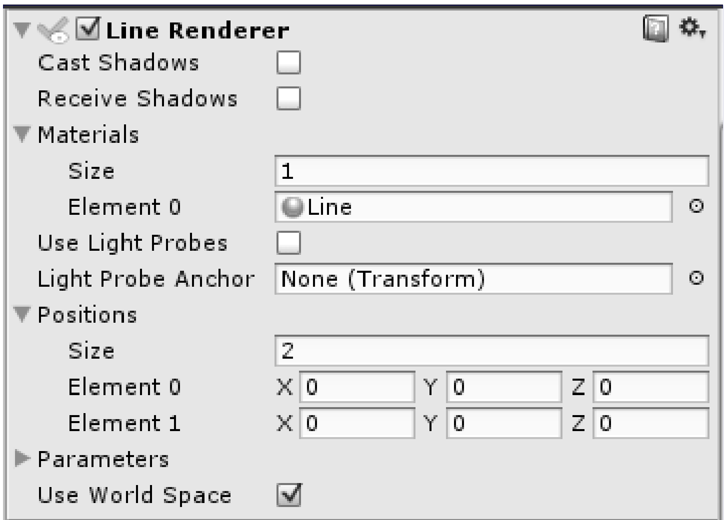
قمنا في الفصل الأول من الوحدة الثالثة بتطبيق نظام مقذوفات بسيط اعتمد حينها على تحريك الأجسام في اتجاه محدد بسرعة ثابتة. في هذا الفصل سنستفيد من إمكانات المحاكي الفيزيائي وتحديدا الأجسام الصلبة وقوة الاندفاع وذلك بهدف بناء نظام مقذوفات أكثر واقعية. ببساطة شديدة يمكن تحويل أي جسم صلب إلى مقذوف بمجرد إضافة قوة اندفاع بمقدار واتجاه محددين. ما سنقوم ببنائه في هذا الفصل هو جسم صلب على شكل كرة يتم قذفها على مجموعة من الصناديق المكدّسة فيما يشبه ميكانيكية اللعب في اللعبة الشهيرة Angry Birds. المشهد الذي سنحتاجه شبيه بالذي في الشكل 64.



الشكل 64: المشهد الخاص بتطبيق المقذوفات

لإطلاق الكرة ينبغي على اللاعب أن يضغط بزر الفأرة الأيسر على الكرة ثم يسحب نحو اليسار، بالمقابل ستقوم اللعبة برسم خط يبدأ من موقع الكرة وينتهي بموقع مؤشر الفأرة. بزيادة طول هذا الخط (أي زيادة المسافة بين مؤشر الفأرة وموقع الكرة) سيزيد مقدار قوة الاندفاع. هذه الميكانيكية البسيطة تسمح للاعب باختيار مقدار واتجاه حركة الإطلاق التي يحتاجها. ملاحظة أخرى مهمّة هي كون المشهد مبنيا باستخدام الأبعاد الثلاثية، إلّا أننا سنقصر الحركة على بعدين فقط. معنى هذا أننا سنقوم بتجميد حركة كافّة الأجسام الصلبة على المحور z. كما أننا سنحتاج لتجميد دوران جميع الأجسام الصلبة على المحورين y و x وحصره فقط على المحور z. نتيجة لذلك علينا أن نتأكد من أن جميع مواقع الأجسام في الفضاء تحمل نفس القيمة للموقع z، وذلك حتى نضمن أنها ستتصادم مع بعضها.

إضافة للمكوّنات الفيزيائية سنستخدم في هذا المثال مكوّنا جديدا هو Line Renderer وسنقوم بإضافته لكائن الكرة. الغرض من هذا المكوّن هو رسم خط في الفضاء ثلاثي الأبعاد يمر بمجموعة من المواقع التي يمكننا تحديدها له. الشكل 65 يبين خصائص هذا المكوّن والذي سنستخدمه لنظهر للاعب مقدار واتجاه الاندفاع الذي هو على وشك تطبيقه على الكرة.



الشكل 65: مكوّن Line Renderer الذي سنستخدمه لعرض اتجاه ومقدار قوة الاندفاع التي ستقذف الكرة

يمكنك استخدام أي خامة تريدها لرسم هذا الخط، حيث استخدمت لهذا المثال خامة تحمل الاسم Line وهي عبارة عن تدريج لوني بين الأزرق والبرتقالي. عدد المواقع التي نحتاجها والتي يتم تحديدها عبر المصفوفة Positions هو 2: نقطة بداية للخط ونقطة نهاية له.سنترك القيم الافتراضية للمواقع كما هي حيث سنقوم لاحقا بتغييرها من خلال البريمج PhysicsProjectile والذي سنقوم أيضا بإضافته لكائن الكرة حتى نتمكن من التحكم بها. هذا البريمج موضح في السرد 54.

```
using UnityEngine;
using System.Collections;

public class PhysicsProjectile : MonoBehaviour {

	//عامل قوة الإطلاق الخاصةبالاندفاع
	public float launchPower = 300;

	//عدد الثواني التي سنبقي خلالها على المقذوف بعد إطلاقه و قبل تدميره
	//-1 = تعني أن يبقى المقذوف دائما ولا يتم تدميره
	public float lifeTime = 7;

	//هل قام اللاعب بالضغط بزر الفأرة على كائن المقذوف؟
	bool mousePressed = false;

	//هل تم إطلاق المقذوف حتى الآن؟
	bool launched = false;

	//الموقع الذي سيتم توليد قوة الاندفاع منه
	Vector3 launchPosition;

	//الخط الذي سيعرض اتجاه ومقدار الإطلاق
	LineRenderer line;

	//مرجع لكاميرا المشهد
	//سنحتاج للكاميرا لحساب موقع الإطلاق مستخدمين موقع مؤشر الفأرة
	Camera cam;

	void Start () {
		//قم بإيجاد كل من مكوّن رسم الخط والكاميرا
		line = GetComponent<LineRenderer>();
		cam = Camera.main;
	}

	void Update () {
		//سنقوم برسم الخط بعد أن يقوم اللاعب بالضغط بزر الفأرة على المقذوف لإطلاقه
		//وقبل أن يعاود إفلات الزر لإطلاق المقذوف
		if(!launched && mousePressed){
			//قم بتوليد شعاع يبدأ من موقع
			//الكاميرا ممتدا إلى داخل الشاشة، ويمر
			//عبر موقع مؤشر الفأرة
			Ray cameraRay = cam.ScreenPointToRay(Input.mousePosition);

			//قم بحساب المسافة بين موقع الكاميرا وموقع المقذوف
```

```
            float dist = Vector3.Distance(
                  cam.transform.position, transform.position);

            //ستكون نقطة الإطلاق بالتالي هي نقطة على الشعاع والتي تبعد
            //عن الكاميرا بنفس مقدار المسافة بين الكاميرا والمقذوف
            launchPosition = cameraRay.GetPoint (dist);

            //الآن علينا أن نحدد نقطتي البداية والنهاية للخط الذي سنقوم برسمه
            //نقطة البداية ستكون في نفس موقع المقذوف
            line.SetPosition(0, transform.position);

            //أما نقطة النهاية فهي نقطة الإطلاق التي قمنا بحسابها
            line.SetPosition(1, launchPosition);

            //أخيرا علينا أن نقوم بتغيير القيمة z الخاصة بنقطة الإطلاق
            //بحيث تصبح مساوية لنفس القيمة في موقع المقذوف
            launchPosition.z = transform.position.z;
         }
      }

      //يتم استدعاء هذه الدّالة عند الضغط بالفأرة على الكائن
      void OnMouseDown(){
         mousePressed = true;
      }

      //يتم استدعاء هذه الدّالة عندما يتم إفلات زر الفأرة
      //في حال إذا كان نفس الزر قد تم ضغطه على الكائن
      void OnMouseUp(){
         //يجب ألا يكون المقذوف قد تم إطلاقه مسبقا
         if(!launched){
            //قم بتغيير قيمة launched إلى true ومن ثم قم بتدمير مكوّن الخط
            launched = true;
            Destroy(line);

            //قم بتدمير الكائن بعد انتهاء مدة الإبقاء عليه
            Destroy(gameObject, lifeTime);

            //قم بتطبيق قوة الاندفاع في اتجاه متناسب
            //مع المسافة بين موقع الإطلاق وبين موقع المقذوف

            Vector3 forceDirection =
                  transform.position - launchPosition;
            forceDirection = forceDirection * launchPower;
            rigidbody.AddForce(forceDirection, ForceMode.Impulse);
         }
      }
}
```

السرد 54: البريمج الخاص بإطلاق المقذوف عن طريق إضافة قوة اندفاع ويتم التحكم به عن طريق الفأرة

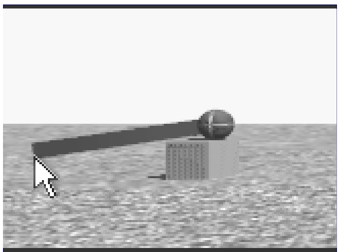
يحتوي البريمج على متغيرين عامين هما launchPower و الذي يحدد مقدار قوة الاندفاع التي سنطبقها لإطلاق المقذوف بعد أن نضربها بالمسافة التي يتحكم بها اللاعب عند تحديد نقطة الإطلاق.

المتغير الآخر هو lifeTime والذي يحدد عدد الثواني التي سنبقي فيها المقذوف في المشهد بعد أن يتم إطلاقه. بالإضافة لهذين المتغيرين لدينا متغيرا الحالة mousePressed والذي نقوم بتغيير قيمته إلى true بمجرد أن يقوم اللاعب بالضغط بالفأرة على المقذوف، إضافة إلى المتغير launched والذي تبقى قيمته false إلى أن يتم إطلاق المقذوف. تكمن أهمية المتغير launched في أنّه يضمن ألا نسمح للاعب بإطلاق المقذوف أكثر من مرة. المتغير الآخر المهم هو launchPosition والذي يمثل الطرف الآخر للخط الذي سنقوم برسمه ابتداء من موقع المقذوف، كما أنّه يحدد مقدار القوة التي يرغب اللاعب بتطبيقها على المقذوف. أخير لدينا مرجعان لكل من كاميرا المشهد ومكوّن رسم الخط Line Renderer الذي قمنا بإضافته على كائن المقذوف. لاحظ هنا استخدامنا للمتغير Camera.main والذي يعطينا مرجعا مباشرا لكاميرا المشهد.

التنفيذ الفعلي لعملية إطلاق المقذوف تتم عبر الدّالتين ()OnMouseDown و ()OnMouseUp، فبعد أن يضغط اللاعب بالفأرة على الكرة ويبدأ بالسحب، تقوم الدّالة ()LateUpdate بتحديث الخط المرسوم عن طريق المكوّن Line Renderer. بما أن الدّالة ()OnMouseDown تُستدعى عند ضغط اللاعب على الفأرة فوق كائن المقذوف، فإنّنا نستخدمها لتغيير قيمة mousePressed إلى true. ما نتوقعه بعد ذلك هو أن يقوم اللاعب بتحريك الفأرة من أجل تحديد اتجاه الإطلاق، بالتالي نقوم في الدّالة ()LateUpdate بتحديث الخط المرسوم بين المقذوف ومؤشر الفأرة، وهو الأمر الذي يجب أن يتم فقط في حال تم الضغط على المقذوف ولم يتم إطلاقه بعد. لنتمكن من تحديد موقع النهاية الأخرى للخط فإننا نحتاج لأن نعرف موقع النقطة في فضاء المشهد التي يغطيها مؤشر الفأرة حاليا. من أجل ذلك نرسم خطا مستقيما (شعاع) ابتداء من موقع الكاميرا ومرورا بمؤشر الفأرة وانطلاقا داخل فضاء المشهد. من أجل إنشاء هذا الشعاع نحتاج لاستدعاء الدّالة ()camera.ScreenPointToRay.

لنتمكن من فهم آلية عمل الدّالة ()ScreenPointToRay وكيف تفيدنا في تحديد نقطة الإطلاق، لنتخيل أن مؤشر الفأرة هو كائن موجود في فضاء المشهد، ويتواجد أمام الكاميرا مباشرة وأقرب إليها من أي كائن آخر. فإذا رسمنا خطا مستقيما يبدأ من موقع الكاميرا مرورا بالمؤشر، فإن امتداد هذا الخط سيكون داخل فضاء المشهد في الاتجاه الذي يراه اللاعب. ما نحتاجه الآن هو نقطة على هذا الخط تكون بعيدة عن الكاميرا بنفس مقدار بعد المقذوف، بالتالي يلزمنا أولا حساب المسافة بين الكاميرا والمقذوف وهي المسافة التي نخزنها في المتغير dist. في هذه الحالة تحديدا تساعدنا الدّالة ()cameraRay.GetPoint. عند استدعاء هذه الدّالة فإننا نزودها بمقدار مسافة معينة، وتعطينا بالمقابل إحداثيات نقطة على الشعاع تبعد بمقدار نفس المسافة عن مركز الشعاع. وبما أن مركز الشعاع هو الكاميرا نفسها، فإنّ هذه النقطة تبعد عن الكاميرا بنفس مسافة dist أي بنفس المسافة بين المقذوف والكاميرا، وهي النقطة التي نخزنها أخيرا في المتغير launchPosition. بقي علينا الآن أن نقوم بتحديث الخط المرسوم عن طريق المكوّن Line Renderer. في كل مرة نستدعي الدّالة ()line.SetPosition نحتاج لتزويدها بمتغيرين، الأول هو ترتيب النقطة على الخط والإحداثي الثاني هو موقع النقطة في فضاء المشهد. في حالتنا هذه لدينا نقطتان: الأولى ستكون في موقع المقذوف نفسه transform.position والثانية ستكون في النقطة التي يقع فوقها مؤشر الفأرة والتي حسبناها سابقا

وهي launchPosition. بعد تحديد هذين النقطتين سيظهر الخط ممتدا بين المقذوف ومؤشر الفأرة كما في الشكل. لاحظ أنه بعد رسم الخط نغير الإحداثي z في الموقع launchPosition ليصبح مساويا لمثيله في موقع المقذوف، مما يجعل اتجاه قوة الإطلاق صحيحا. لكن هذا الموقع لا يصلح لرسم الخط لأنه سيظهر طرف الخط بعيدا عن المؤشر.



الشكل 66: الخط الممتد بين موقع المقذوف وموقع مؤشر الفأرة

بقي الآن أن نتعامل مع أهم حدث وهو إفلات اللاعب لزر الفأرة مما يؤدي لإطلاق المقذوف. هذا الحدث يتم التعامل معه عبر الدّالة ()OnMouseUp والتي تستدعى مرة واحدة فقط. إذا كانت قيمة متغير الحالة launched هي false، فهذا يعني أن المقذوف لم يتم إطلاقه بعد، لذا فأول ما نقوم به هو منع إعادة الإطلاق وذلك بتغيير قيمته إلى true. قبل أن نضيف قوة الاندفاع علينا أولا أن نقوم بإزالة الخط وذلك عن طريق تدمير المكوّن باستدعاء الدّالة ()Destroy تماما كما ندمر الكائن. بعدها نقوم أيضا باستدعاء ()Destroy لكن هذه المرة مع تأخير بمقدار lifeTime وذلك من أجل تدمير الكائن بعد انقضاء المدة المحددة. بعدها علينا أن نحسب اتجاه قوة الإطلاق التي سنستخدمها وهو بطبيعة الحال المتجه الممتد من موقع الإطلاق إلى موقع المقذوف والذي نحسبه بطرح الأول من الثاني ونخزنه في forceDirection. بطبيعة الحال فإن هذا المتجه يكون أطول أي مقدار أكبر كلما ابتعدت نقطة الإطلاق عن موقع المقذوف. قبل إضافة القوة للمقذوف نقوم بضرب المتجه بقيمة launchPower لتكبيرها بالقدر اللازم.

البريمج الآخر الذي سنحتاج إليه هو بريمج بسيط يقوم بإضافة مقذوف كلما أطلقنا آخر. هذا البريمج هو ProjectileGenerator وموضح في السرد 55. الشكل 67 يوضح المقذوف لحظة اصطدامه بالصناديق، ويمكنك مشاهدة النتيجة النهائية في المشهد scene18 في المشروع المرفق.

```
using UnityEngine;
using System.Collections;

public class ProjectileGenerator : MonoBehaviour {

	//قالب المقذوف الذي سنقوم بتوليده
	public GameObject projectile;
```

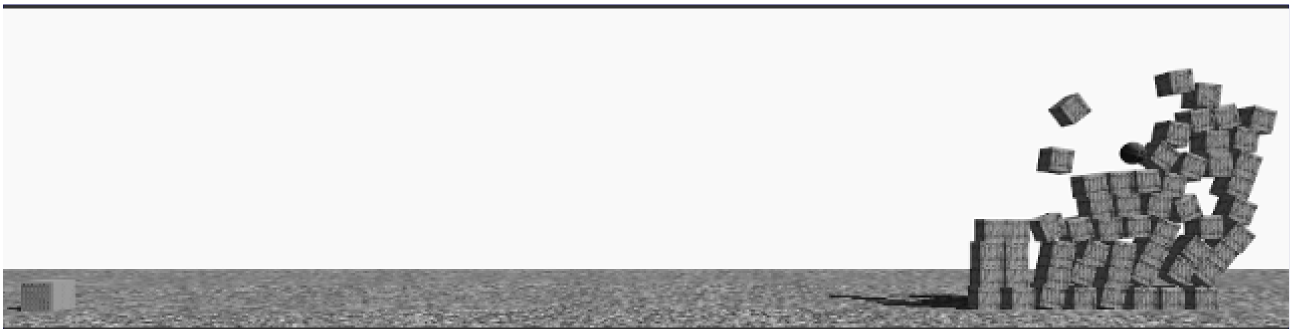
```
    void Start () {
        //قم بمحاولة توليد المقذوف مرة كل ثانية
        InvokeRepeating("Generate", 0, 1);
    }

    void Update () {

    }

    void Generate(){
        //إذا لم يكن هناك أي مقذوف في المشهد قم بتوليد واحد جديد
        PhysicsProjectile[] prjectiles =
            FindObjectsOfType<PhysicsProjectile>();
        if(prjectiles.Length == 0){
            Instantiate(projectile,
                        transform.position,
                        transform.rotation);
        }
    }
}
```

السرد 55: البريمج الخاص بتوليد المقذوفات

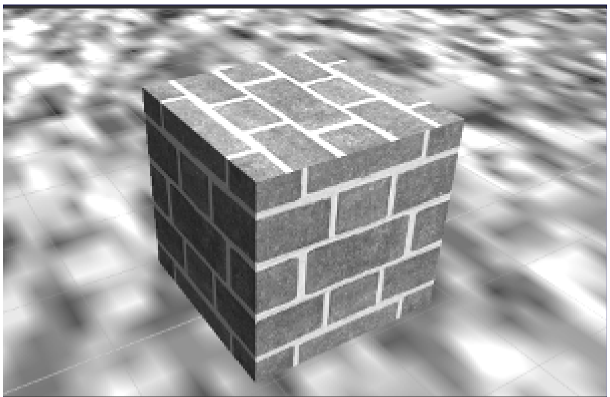


الشكل 67: المقذوف لحظة إطلاقه واصطدامه بالصناديق

الفصل السادس: الانفجارات وهدم المنشآت

هذا الفصل ذو أهمية خاصة بتطوير ألعاب الأكشن، والتي تعتبر فيها الانفجارات وتهدّم أجزاء من بيئة اللعب من الميكانيكيات الرئيسية. لحسن الحظ يسهل علينا وجود المحاكي الفيزيائي صناعة انفجارات قريبة من الواقع وذلك من خلال القدرة على دفع الأجسام بعيدا عن مركز الانفجار باستخدام القوى الفيزيائية. إضافة لذلك يسهل علينا المحاكي الفيزيائي عملية صناعة منشآت قابلة للهدم وتتأثر بالانفجارات. سنقوم في هذال الفصل بتصميم مبنى بسيط يتألّف من مجموعة من الوحدات البنائية. ما يميز هذه الوحدات هو أنها منفصلة تماما عن بعضها وتتأثر بالانفجارات التي تقع بالقرب منها، مما يوصلنا أخيرا إلى مبنى يمكن هدمه بشكل جزئي أو كلي بفعل الانفجارات. سنقوم أيضا بإضافة ميزة فريدة لكل وحدة بنائية وهي القدرة على العودة لمكانها الأصلي، مما يجعل المبنى كله قابلا لإعادة

البناء. الخطوة الأولى ستكون عمل القالب الخاص بالوحدة البنائية ومن ثم استخدام عدد كبير من النسخ لإنشاء المبنى. لعمل القالب سنحتاج لمكعب بالحجم الافتراضي يحمل إكساء على شكل لبنات البناء كما في الشكل 68.



الشكل 68: مكعب مع إكساء على شكل لبنات سنستخدمه كوحدة بنائية للمبنى

أول خطوة لبناء القالب المطلوب هي إضافة مكوّن الجسم الصلب إلى المكعب، ومن ثم علينا أن نقوم بتجميد هذا الجسم الصلب عبر البريمج Destructible الموضّح في السرد 56. الهدف من التجميد هو أن تبقى كل وحدة بنائية ثابتة في مكانها في الوضع الطبيعي. هذا التجميد سنقوم بفكّه في حال أثّرت قوة خارجية بمقدار كاف على الوحدة البنائية وحركتها من مكانها. في حالتنا هذه القوّة الخارجية المفترضة هي قوة الانفجار.

```
using UnityEngine;
//سنحتاج لاستيراد هذه المكتبة
using System.Collections.Generic;

public class Destructible : MonoBehaviour {

	//سنقوم بإجراء مسح عن طريق شعاع ابتداء من مركز هذه الوحدة
	//وانطلاقا في الاتجاه المحدد عبر هذا المتجه
	public Vector3 scanDirection;

	//قائمة بالوحدات البنائية التي تعتمد في ثباتها على هذه الوحدة
	List<Destructible> dependents = new List<Destructible>();

	//متغير لتخزين قيم حرية الحركة الأصلية الخاصّة بالجسم الصلب قبل تجميده
	RigidbodyConstraints original;

	void Start () {
		//قم بالبحث عن وحدة بنائية أخرى مستخدما اتجاه المسح المحدد سلفا
		//إذا ما تم العثور على وحدة ما فإننا نضيف هذه الوحدة إلى قائمة الوحدات المعتمدة عليها
		RaycastHit hit;
```

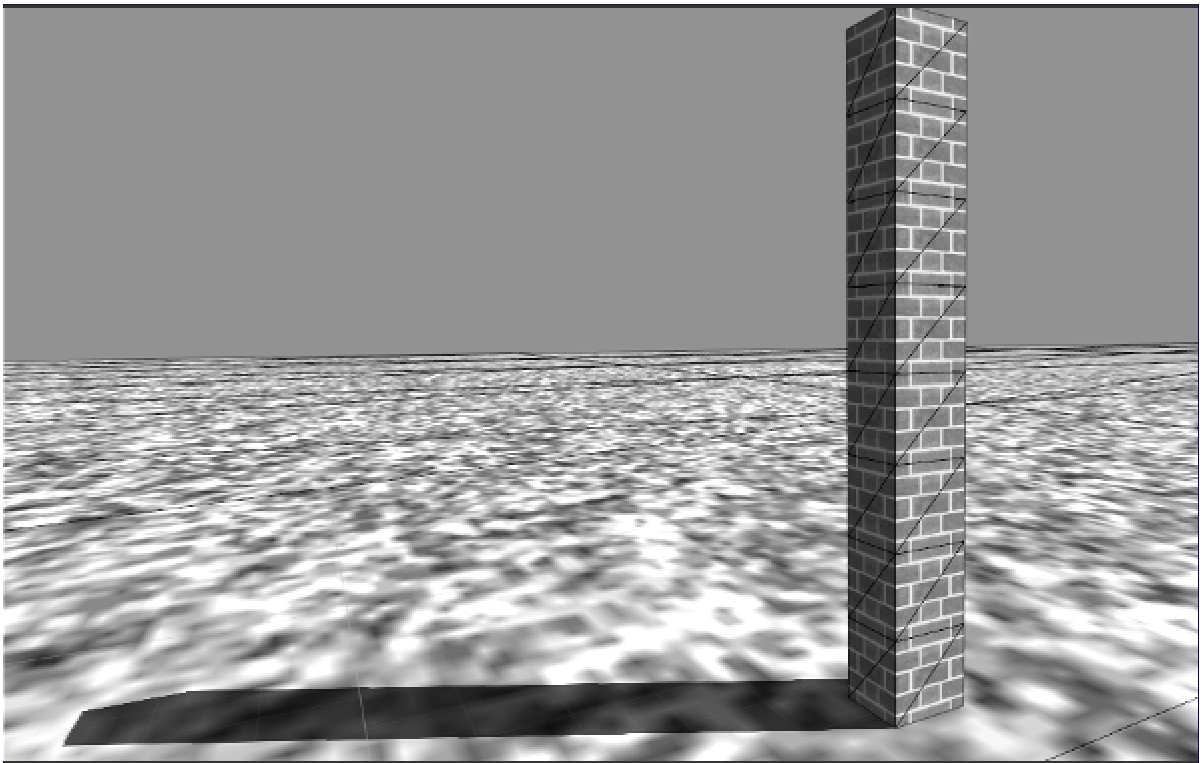
```
21.             Ray scanRay = new Ray(transform.position, scanDirection);
22.
23.           if(Physics.Raycast(scanRay, out hit, scanDirection.magnitude)){
24.
25.                 Destructible dependency =
26.                       hit.transform.GetComponent<Destructible>();
27.
28.                 if(dependency != null){
29.                       dependency.dependents.Add(this);
30.                 }
31.           }
32.
33.           //قم بتخزين قيم حرية الحركة والدوران الأصلية ومن ثم قم بتجميد الكائن
34.           original = rigidbody.constraints;
35.           rigidbody.constraints = RigidbodyConstraints.FreezeAll;
36.     }
37.
38.     void Update () {
39.
40.     }
41.
42.     //قم بهدم هذه الوحدة البنائية وذلك باستعادة قيم حرية الحركة الأصلية الخاصّة بها
43.     public void Destruct(){
44.           rigidbody.constraints = original;
45.           //قم باستدعاء مؤجّل للدّالة Destruct() من جميع الوحدات التي تعتمد على هذه الوحدة
46.           foreach(Destructible dependent in dependents){
47.                 if(dependent != null){
48.                       float time = Random.Range(0.0f, 0.01f);
49.                       dependent.Invoke("Destruct", time);
50.                 }
51.           }
52.           //قم بإعلام البريمجات الأخرى بحدوث عملية الهدم لهذه الوحدة
53.           SendMessage("OnDestruction",
54.                             SendMessageOptions.DontRequireReceiver);
55.     }
56. }
```

السرد 56: البريمج الخاص بالوحدات البنائية للمبنى القابل للتدمير

قبل الخوض في تفاصيل البريمج لابد من التطرق لكلمة using التي استخدمناها في السطر الثالث لإضافة مكتبة جديدة لهذا البريمج وهي مكتبة System.Collections.Generic. ماهية المكتبة ليست مهمة بقدر أهمية معرفتنا لكيفية استخدامها. بعد استيراد هذه المكتبة يصبح من الممكن أن نقوم بتعريف قائمة List تحتوي على عناصر من نوع Distructible وذلك باستخدام النوع List<Distructible>. القوائم تشبه المصفوفات من حيث أنها تحتوي على مجموعة من العناصر ذات نفس النوع، إلا أنها أكثر مرونة ويمكننا بسهولة أن نضيف ونحذف العناصر منها ولا يلزمنا تحديد عدد العناصر مسبقا.

المتغيران الأكثر أهمية في هذا البريمج هما scanDirection و dependents. حتى ندرك أهمية هذين المتغيرين علينا أولا أن نفهم آلية عمل الوحدات البنائية والعلاقات بينها. عند إنشاء أي مبنى باستخدام

هذه الوحدات البنائية، فأنّ ما علينا فعله هو توزيع هذه الوحدات بصورة معينة أفقيا وعموديا حتى تعطي الشكل المطلوب. مثالا على ذلك، عندما نحتاج لإنشاء عمود كالذي في الشكل 69، فإنّ ما علينا فعله هو ترتيب عدد من الوحدات البنائية فوق بعضها البعض.



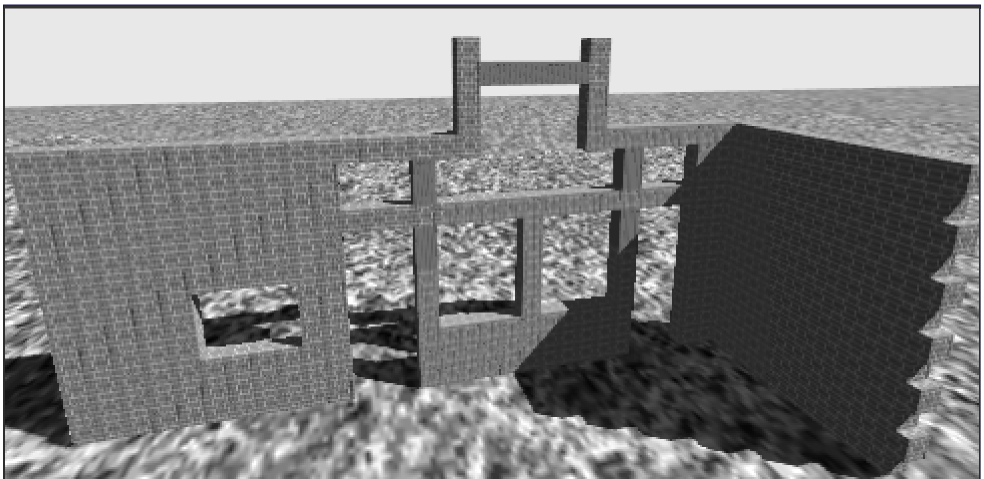
الشكل 69: عمود تم إنشاؤه باستخدام الوحدات البنائية

تبعا للعلاقة المنطقية بين أجزاء العمود، فإننا نتوقع أنهياره بشكل تام عند تهدم الوحدة البنائية في الأسفل (يستثنى من ذلك المكعبات الطائرة في الهواء كالتي في لعبة Super Mario). لأجل ذلك علينا إعلام المحاكي الفيزيائي بأن كل وحدة بنائية تعتمد على تلك التي أسفل منها وبالتالي تتهدم إذا تهدم ما تحتها. من أجل هذا الغرض نستخدم المتجه scanDirection والذي يبدأ من مركز الوحدة البنائية متجها في اتجاه معين. في حالة العمود مثلا يجب أن يتجه نحو الأسفل ويكون بطول يزيد عن 0.5 وهي المسافة بين مركز المكعب وسطحه الأسفل. قبل أن ننتقل للحديث عن آلية عمل scanDirection من المهم أن نفهم طبيعة العلاقات بين الوحدات البنائية. لكل وحدة من هذه الوحدات قائمة تحتوي على مجموعة من الوحدات الأخرى التي تبني عليها، فعندما يتم تهديم الوحدة الأساس فإن كافّة الوحدات التي تعتمد عليها والموجودة في قائمة dependents يجب أن تتهدم كذلك.

عندما يتم استدعاء الدّالّة ()Start حين بداية تشغيل اللعبة، تقوم كل وحدة بنائية ببث شعاع بالاتجاه والطول المحددين عبر scanDirection، في محاولة لإيجاد وحدة بنائية أخرى تعتمد عليها. بطبيعة الحال فإنّ الوحدة البنائية يجب أيضا أن تحتوي على البريمج Destructible وهذا ما يتم التأكد منه في السطر

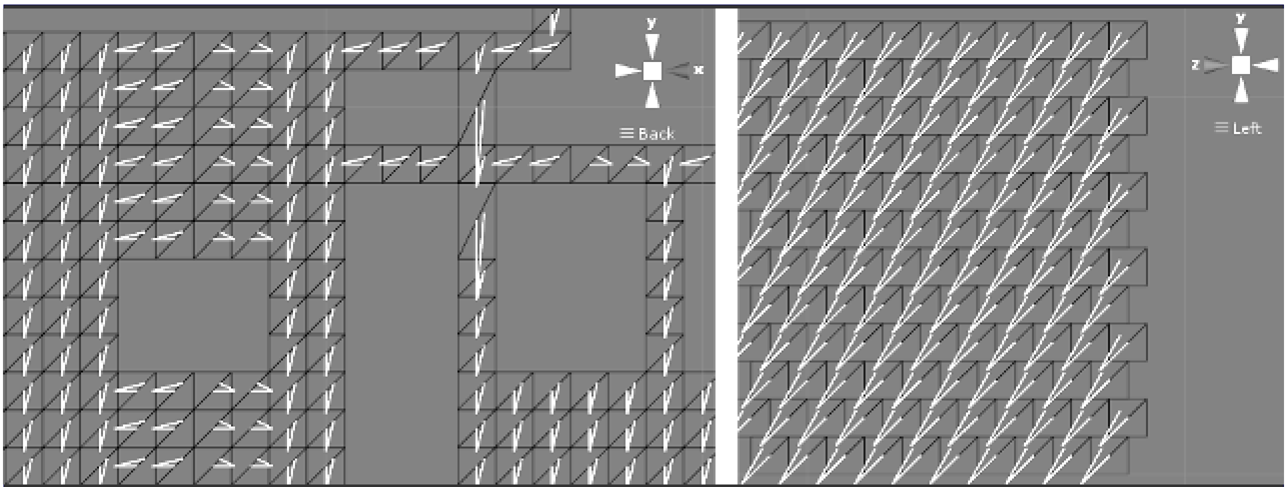
28. إذا وُجدت الوحدة المطلوبة فإنّ الوحدة الحالية (أي التي قامت ببث الشعاع) تضيف نفسها إلى قائمة dependents في الوحدة الأخرى التي اصطدم بها الشعاع. في حالو العمود الموضح في الشكل 69 فإنّ scanDirection يجب أن يحمل القيمة (0, -0.6, 0) مما يجعل شعاع المسح يتجه نحو الأسفل. نتيجة لذلك فإنّ كل وحدة بنائية في العمود ستضيف نفسها إلى قائمة dependents في الوحدة الأسفل منها، بينما لن تجد الوحدة الواقعة في قاعدة العمود أي وحدة أخرى تعتمد عليها.

بعد الانتهاء من بناء العلاقات بين الوحدات البنائية يجب أن يتم تجميد هذه الوحدات حتى لا تتأثر بالقوى الفيزيائية من حولها. تنفيذ التجميد يتم عبر تخزين القيم الأصلية للمتغير rigidbody.constraints في المتغير original ومن ثم تغيير قيمته إلى RigidbodyConstraints.FreezeAll. معنى ذلك أن كلا من الموقع والدوران الخاصين بالوحدة البنائية لن يتغيرا إلى أن نسمح بذلك مرة أخرى. بالتالي ستبقى الوحدة البنائية جامدة في مكانها إلى أن يتم استدعاء الدّالّة ()Destruct أو استقبال رسالة تحمل الاسم Destruct. عند استدعاء هذه الدّالّة نقوم بالسماح للوحدة بالتحرك تحت تأثير المحاكي الفيزيائي وذلك باسترجاع القيم الأصلية الخاصة بـ rigidbody.constraints والتي سبق تخزينها في original. بعد ذلك لا بد من نشر عملية الهدم إلى كل الوحدات البنائية التي تعتمد على هذه الوحدة وذلك عن طريق استدعاء الدّالّة ()Destruct من جميع الوحدات البنائية الموجودة في القائمة dependents. من الأفضل عند استدعاء هذه الدّالّة أن نضيف تأخيرا زمنيا بسيطا يعطي انطباعا عن طبيعة العلاقة بين الوحدات البنائية، بحيث يرى اللاعب الوحدة الأصلية تهدم أولا ثم تتبعها الوحدات الوحدات المعتمدة عليها على شكل انهيار متدرج للبناء. من المفيد أيضا إعلام أية بريمجات أخرى بحصول الهدم وذلك عن طريق إرسال الرسالة OnDestruction مما يمكننا من تنفيذ أية تأثيرات أخرى تتعلق بالهدم مثل تشغيل صوت أو إحداث غبار. الشكل 70 يظهر مبنى أكثر تعقيدا تم بناؤه باستخدام عدد أكبر من الوحدات البنائية.


الشكل 70: مبنى تم إنشاؤه بشكل كامل من وحدات بنائية قابلة للهدم

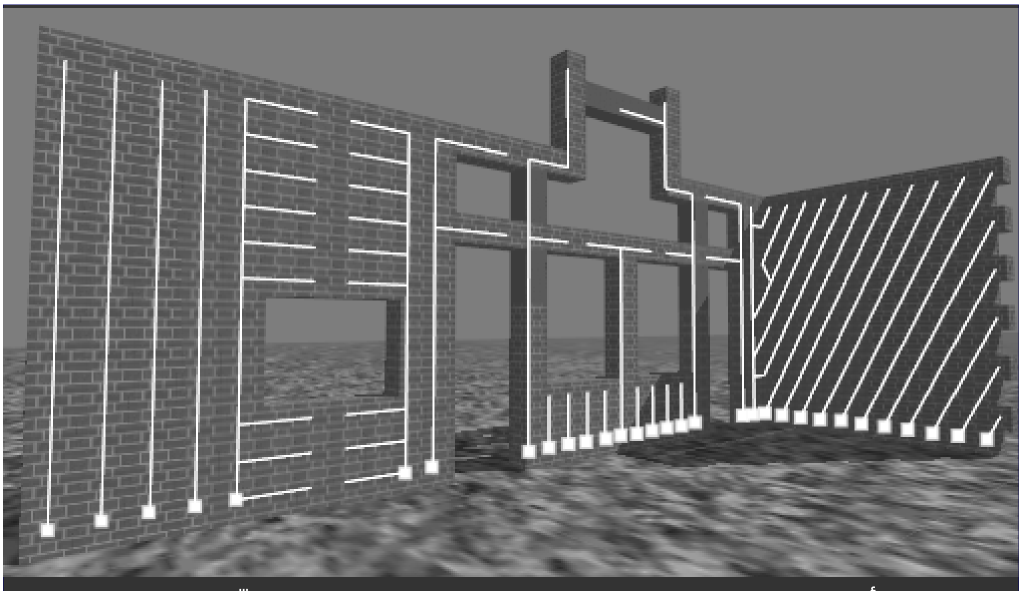
تحديد قيمة scanDirection الخاصّة بكل وحدة بنائية يعتمد بشكل كامل على الوحدات المجاورة لها. الشكل 71 يظهر نفس المبنى مع إضافة أسهم تبين متجهات scanDirection. كل واحد من هذه

الأسهم يبدأ من منتصف الوحدة ويؤشر رأسه إلى اتجاه المسح حيث الوحدة التي يعتمد عليها.



الشكل 71: تغيير اتجاه المسح تبعا لتغير ترتيب الوحدات البنائية. يسارا: يتم المسح للبحث عن وحدات يعتمد عليها في اتجاه أفقي لليمن أو اليسار أو في اتجاه عمودي نحو الأسفل. يمينا: يتم المسح باتجاه واحد فقط هو أسفل يسار الوحدة

نتيجة لعمليات المسح فإننا نحصل في النهاية على علاقات أشبه بالأشجار بين الوحدات البنائية، بحيث تعتمد الفروع في هذه الأشجار على الجذور وتتهدم إذا تهدمت جذورها. الشكل 72 يوضح مثالا على هذه الأشجار.



الشكل 72: أشجار العلاقات بين الوحدات البنائية. عند تهدم الجذر فإنّ كافّة الفروع المتصلة به يجب أن تتهدم كذلك. الجذور في الصورة تظهر على شكل مربعات صغيرة في الأسفل

نحتاج الآن لتنفيذ بعض الانفجارات حتى نقوم بتجربة نظام التهدم الذي بنيناه للتو. سنستعمل لتحقيق

ذلك بريمجا يقوم بتوليد انفجار في أي نقطة نضغط عليها بزر الفأرة الأيسر. البريمج MouseExploder والموضح في السرد 57 هو البريمج الثاني الذي نحتاج لإضافته لقالب الوحدة البنائية.

```
using UnityEngine;
using System.Collections;

public class MouseExploder : MonoBehaviour {

	//مقدار قوة الانفجار
	public float explosionForce = 40000;

	//نصف قطر محيط تأثير الانفجار
	public float explosionRadius = 5;

	//موقع الانفجار
	//هذا الموقع نسبي إلى موقع الجسم أو الوحدة البنائية التي تتعرض للانفجار
	//ممثلا بإحداثيات فضاء المشهد
	public Vector3 explosionPosition = new Vector3(-1, 0, -1);

	void Start () {

	}

	void Update () {

	}

	//تنفيذ الانفجار بمجرد الضغط على زر الفأرة
	void OnMouseDown(){
		//ابحث عن كافّة الأجسام الصلبة في المشهد
		Rigidbody[] allBodies = FindObjectsOfType<Rigidbody>();

		//قم بحساب موقع الانفجار
		Vector3 explosionPos = transform.position;
		explosionPos += explosionPosition;

		//قم بإيجاد كافّة الأجسام الصلبة ضمن محيط الانفجار ومن ثم
		//أرسل الرسالة Destruct لتنفيذ التدمير في حال وجود وحدة بنائية
		//أخيرا قم بإضافة قوة الانفجار للجسم الصلب
		foreach(Rigidbody body in allBodies){
			float dist =
				Vector3.Distance(
					body.transform.position,
					explosionPos);

			if(dist < explosionRadius){
				body.SendMessage("Destruct",
					SendMessageOptions.DontRequireReceiver);

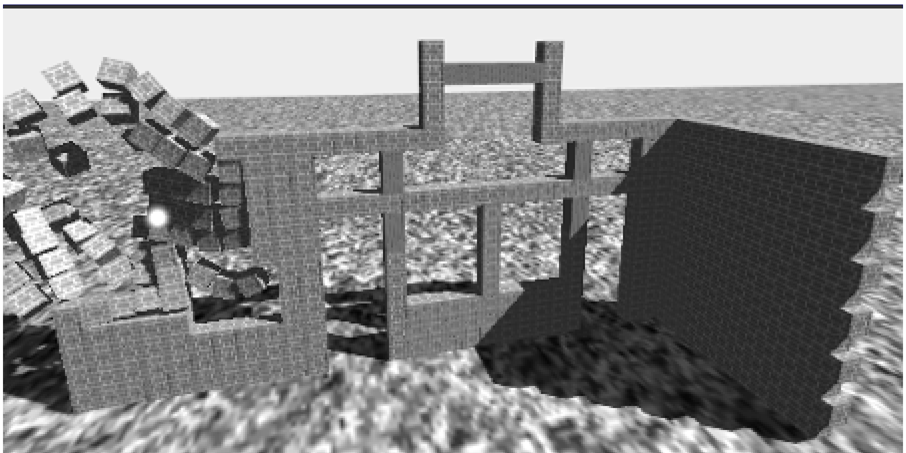
				body.AddExplosionForce(
					explosionForce, //قوة الانفجار
					explosionPos, //موقع الانفجار
					explosionRadius);//نصف قطر التأثير
```

```
51.                    }
52.             }
53.       }
54. }
```

السرد 57: بريمج يقوم بإنشاء انفجار في النقطة التي يتم الضغط عليها بزر الفأرة الأيسر

لعل أول الأشياء التي تلفت النظر في هذا البريمج هو القيمة العالية لمقدار الانفجار explosionForce، وهو أمر متوقع كوننا نتحدث عن قوة تفجيرية يجب أن تمتلك القدرة على هدم مبنى ولو بشكل جزئي. يمكننا تحديد محيط التأثير الخاص بكل انفجار عن طريق تحديد نصف قطر هذا المحيط عبر المتغير exlosionRadius، والذي يحصر تأثير الانفجار على الأجسام الواقعة ضمن هذه المسافة. يقوم هذا البريمج بإحداث انفجار في موقع الوحدة التي يتم الضغط عليها بزر الفأرة الأيسر، بيد أن مركز الانفجار يجب أن يُزاح قليلا عن موقع الوحدة حتى يظهر تأثير الانفجار بشكل واضح. هذه الإزاحة يمثلها المتغير explosionPosition وهي إزاحة في فضاء المشهد نسبية إلى موقع الوحدة التي تم الضغط عليها.

تقوم الدّالّة ()OnMouseClick باستقبال الضغط بزر الفأرة على الوحدة البنائية ومن ثم تقوم بإضافة قوة انفجار لجميع الأجسام الصلبة المحيطة بموقع الانفجار. كخطوة أولى يتم البحث عن كافّة الأجسام الصلبة الموجودة في المشهد وتخزينها في المصفوفة allBodies. بعد ذلك يتم حساب موقع الانفجار explosionPos عن طريق إضافة الإزاحة explosionPosition إلى موقع الوحدة التي تم الضغط عليها. تقوم الحلقة for الموجودة في السطر 37 بالمرور على جميع الأجسام الصلبة وحساب المسافة بينها وبين موقع الانفجار، فإذا كانت المسافة أقل من نصف قطر محيط التأثير explosionRadius يتم إرسال الرسالة Destruct للجسم الصلب وذلك بهدف فك التجميد في حال وجوده، حيث أنّ أي قوة انفجار لن يكون لها أي تأثير ما لم يتم فك تجميد الجسم الصلب أولا. الشكل 73 يظهر تأثير أحد الانفجارات على المبنى. لاحظ أن الانفجار ذو تأثير فيزيائي فقط حيث لم نقم بإضافة أي مؤثرات رسومية له كالنار والدخان.



الشكل 73: تأثير الانفجار على المبنى القابل للهدم. لاحظ اندفاع الوحدات البنائية بعيدا عن مركز الانفجار المشار إليه ببقعة بيضاء

لجعل هذا المثال أكثر متعة سأقوم بإضافة خاصية أخرى وهي إمكانية إعادة تشكيل المبنى على صورته الأصلية بعد أن يهدم جزئيا أو كليا. تطبيق فكرة كهذه يمكن أن يتم ببساطة عن طريق تخزين الموقع والدوران الأصليين لكل وحدة بنائية عند بداية التشغيل، ومن ثم إعادة هذه الوحدات لمكانها بشكل انسيابي عند الضغط على مفتاح ما وليكن مفتاح المسافة مثلا. البريمج Returner يقوم بتنفيذ هذه المهمّة عند إضافته لقالب الوحدة البنائية. هذا البريمج موضّح في السرد 58.

```
using UnityEngine;
using System.Collections;

public class Returner : MonoBehaviour {

	//الموقع الأصلي للوحدة البنائية
	Vector3 originalPos;
	//الدوران الأصلي للوحدة البنائية
	Quaternion originalRot;
	//هل تتحرك الوحدة حاليا نحو موقعها الأصلي؟
	bool returning = false;

	void Start () {
		//قم بحفظ الموقع والدوران الأصليين
		originalPos = transform.position;
		originalRot = transform.rotation;
	}

	void Update () {

		//قم بتنفيذ العودة عند الضغط على مفتاح المسافة
		if(Input.GetKeyDown(KeyCode.Space)){
			Return();
		}

		if(returning){
			//علينا أثناء عملية إعادة الوحدة لموقعها الأصلي أن نقوم بتجميد
			//كافّة الحركات الممكنة حتى نمنع أي تأثير فيزيائي خارجي من إعاقة عملية الإرجاع
			if(rigidbody.constraints !=
				RigidbodyConstraints.FreezeAll){
				//قم بتصفير أي سرعات خطية أو دورانية
				//حتى يتوقف الكائن تماما عن الحركة الفيزيائية
				rigidbody.velocity = Vector3.zero;
				rigidbody.angularVelocity = Vector3.zero;
				//قم الآن بتجميد الحركة والدوران
				rigidbody.constraints =
					RigidbodyConstraints.FreezeAll;
			}

			//قم بشكل سلس بإعادة كل من الموقع والدوران
			//إلى قيمهما الأصلية
			transform.position =
				Vector3.Lerp(transform.position, //من
							originalPos, //إلى
```

```
45.                                          Time.deltaTime * 3);//مقدار الحركة
46.
47.                     transform.rotation =
48.                           Quaternion.Lerp(
49.                                          transform.rotation, //من
50.                                          originalRot, //إلى
51.                                          Time.deltaTime * 3);//مقدار الحركة
52.
53.                     //إذا كان الجسم قريبا جدا من موقعه الأصلي قم بإعادته إلى القيمة الأصلية مباشرة
54.                     //ومن ثم غيّر قيمة returning إلى false
55.                     float remaining =
56.                          Vector3.Distance(transform.position, originalPos);
57.                     if(remaining < 0.01f){
58.                           transform.position = originalPos;
59.                           transform.rotation = originalRot;
60.                           returning = false;
61.                     }
62.               }
63.         }
64.
65.         //قم بإعادة الوحدة البنائية لموقعها الأصلي
66.         public void Return(){
67.               returning = true;
68.         }
69. }
```

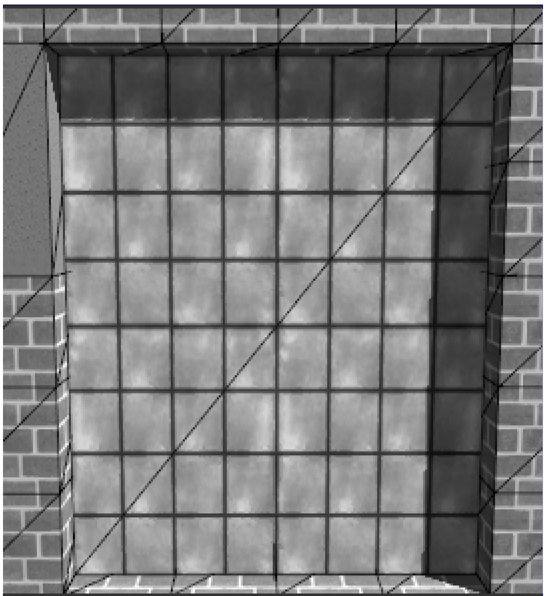
السرد 58: البريمج الخاص بإرجاع الوحدة البنائية لموقعها الأصلي قبل الهدم

ما يقوم به هذا البريمج ابتداء هو تخزين كل من الموقع والدوران الأصليين للكائن وذلك في المتغيرين originalPos و originalRot، وهذا الأخير يحمل النوع Quaternion الخاص بتخزين ومعالجة قيم الدوران. أثناء عملية التحديث نقوم في الدّالّة Update() بفحص حالة مفتاح المسافة، فإذا كان المفتاح مضغوطا فإننا نقوم باستدعاء الدّالّة Return() والتي تقوم بدورها بتغيير قيمة متغير الحالة returning إلى true وذلك حتى يسمح للدّالّة Update() بتحريك الكائن بشكل تدريجي نحو كل من originalPos و originalRot. قبل تنفيذ التحريك والتدوير علينا أن نقوم بتصفير كافّة السرعات الخطية أو الدورانية على الجسم الصلب ومن ثم تجميده حتى لا يتأثر بأي قوى فيزيائية (الأسطر 29 إلى 38). بعد ذلك نستخدم الدّالّتين Vector3.Lerp() و Quaternion.Lerp() لإرجاع الوحدة البنائية بشكل تدريجي وسلس. بضربنا لقيمة Time.deltaTime في 3 نضمن سرعة أكبر في إعادة الوحدة لمكانها. بعد كل عملية تحريك نفحص المسافة بين موقع الوحدة الحالي وموقعها الأصلي، فإذا كانت هذه المسافة أقل من 0.01 فيمكننا حينها أن نعيد الموقع والدوران مباشرة لقيمهما الأصلية. يمكن تجربة النتيجة النهائية في المشهد scene19 في المشروع المرفق.

الفصل السابع: الأجسام القابلة للكسر

تعلمنا في الفصل السابق كيفية إنشاء مبنى قابل للهدم مستخدمين وحدات بنائية منفصلة، وبرغم

كون هذه الطريقة فعّالة في حالة المنشآت والأجسام كبيرة الحجم، إلّا أنها لا تصلح لكل جسم قابل للكسر نريد إنشاءه، ولهذه الأجسام الأصغر حجما نحتاج لحل آخر سنتعلمه في هذا الفصل إن شاء الله. خذ مثلا الصناديق التي تنتشر في كثير من الألعاب والتي يعمد اللاعب إلى فتحها أو كسرها ليستخرج ما بداخلها من ذخيرة أو أدوات أو أي شيء آخر ينفعه خلال اللعبة، أو النوافذ الزجاجية التي تتحطم بسهولة حين اصطدام أي جسم آخر بها. هذا النوع من الكسر يكون عادة غير قابل لإعادة البناء وبالتالي لا يمكن استرجاع الجسم الأصلي أو جزء منه. الفكرة التي سنستخدمها من أجل هذا النوع من الكسر تقوم على حذف الجسم الأصلي من المشهد واستبداله بأجزاء صغيرة تحمل نفس الإكساء، وهذه الأجسام الصغيرة يمكن عملها باستخدام القوالب. لنأخذ مثلا المبنى الذي قمنا بعمله في الفصل السابق، والذي يحتوي على فتحات يمكن تغطيتها بنوافذ زجاجية قابلة للكسر. هذه النوافذ ستكون على شكل قوالب وستكون شبيهة لما في الشكل 74.



الشكل 74: لوح زجاجي للاستخدام كنافذة للمبنى

ما علينا الآن هو إضافة بريمج Destructible لقالب النافذة حتى يصبح كأي وحدة بنائية أخرى في المبنى من ناحية تأثره بقوى الانفجارات. البريمج الآخر الذي نحتاجه هو البريمج Breakable والموضح في السرد 59، ومهمته هي تحويل هذه النافذة إلى جسم قابل للكسر.

```
using UnityEngine;
using System.Collections;

public class Breakable : MonoBehaviour {

        //الأجسام التي سيتم إنشاؤها عند الكسر (قطع الحُطام)
        public GameObject[] pieces;

```

```
	//كم من المرات سيتم تكرار كل قطعة حطام؟
	public int pieceCopies = 5;

	//كم هي القوة التي يحتاجها هذا الجسم لينكسر
	public float explosionPower = 10;

	//كم من الوقت ستبقى قطع الحطام موجودة؟
	public float pieceLifetime = 10;

	void Start () {

	}

	void Update () {

	}

	//قم بكسر الجسم
	public void Break(){
		//قم بإنشاء قطع الحطام تبعا لعددها المحدد
		foreach(GameObject piecePrefab in pieces){
			for(int i = 0; i < pieceCopies; i++){
				GameObject piece =
					(GameObject)Instantiate(piecePrefab);

				//قم باختيار موقع عشوائي للإنشاء
				//يجب أن يكون الموقع ضمن حدود الجسم الأصلي
				Vector3 borders = transform.localScale;
				Vector3 randPos;

				randPos.x =
				Random.Range(-borders.x * 0.5f, borders.x * 0.5f);

				randPos.y =
				Random.Range(-borders.y * 0.5f, borders.y * 0.5f);

				randPos.z =
				Random.Range(-borders.z * 0.5f, borders.z * 0.5f);

				//قم بوضع قطعة الحطام في الموقع العشوائي
				piece.transform.position =
					transform.position + randPos;

				Vector3 explosionPos = transform.position;
				float explosionRad = transform.localScale.magnitude;

				piece.rigidbody.AddExplosionForce(explosionPower,
									explosionPos,
									explosionRad);

				//إن وُجد وقت محدد للإبقاء على قطع الحطام
				//قم بتدمير القطع بعد هذا الوقت
				if(pieceLifetime > 0){
					Destroy(piece, pieceLifetime);
				}
			}
```

```
            }

            //قم بإعلام البريمجات الأخرى بأن هذا الجسم تم كسره
          SendMessage("OnBreak", SendMessageOptions.DontRequireReceiver);

            //قم أخيرا بتدمير الجسم وحذفه من المشهد
            Destroy(gameObject);

        }
}
```

السرد 59: البريمج الخاص بالأجسام القابلة للكسر

بما أننا سنعمل على استبدال الكائن الأصلي بمجموعة من قطع الحطام عند تكسره، يلزمنا أن نقوم بتحديد هذه القطع وتزويدها عن طريق المصفوفة pieces. يمكننا كذلك عن طريق المتغير pieceCopies أن نضبط عدد النسخ التي سنقوم بإنشائها من كل قالب قطعة حطام في المصفوفة pieces. معنى ذلك أنه لو كان عندك – مثلا – قالبان لقطعتي حطام في المصفوفة pieces وقمت بضبط قيمة pieceCopies على 5، فإنك ستحصل على 10 قطع حطام عند تكسير الكائن. عند تكسير الكائن فإنه يصدر قوة انفجار يمكن تحديدها عن طريق المتغير explosionPower، مما يؤثر على قوة تنافر عدد قطع الحطام بعيدا عن الجسم. جدير بالذكر أن قوة الانفجار هذه ستطبق فقط على قطع الحطام لتنثرها بعيدا عن موقع الجسم الأصلي ولكنها لا تؤثر على أي من الكائنات الأخرى المحيطة. يمكننا أخيراً أن نقوم بتحديد الفترة الزمنية التي سنبقي فيها على قطع الحطام قبل حذفها من المشهد، وذلك عبر المتغير pieceLifeTime. بضبط قيمة هذا المتغير على صفر أو أقل يمكنك الإبقاء على قطع الحطام في المشهد بشكل دائم.

في اللحظة التي نقرر فيها أن هذا الكائن يجب أن يتكسر نقوم باستدعاء الدّالة ()Break والتي تقوم بثلاث مهام: إنشاء نسخ من قطع الحطام، وإبلاغ البريمجات الأخرى على الكائن بأنّ الكائن تم كسره وذلك عن طريق إرسال الرسالة OnBreak، ومن ثم تقوم أخيرا بحذف الكائن من المشهد. الجزء المثير للاهتمام هنا هو الحلقتان المتداخلتان foreach و for، حيث تقوم الحلقة الخارجية بالمرور على جميع القوالب الموجودة في المصفوفة pieces وتقوم بتنفيذ الداخلية والتي تتكرر بعدد يساوي pieceCopies بهدف إنشاء العدد المطلوب من قطع الحطام من كل قالب. عند إنشاء كل قطعة حطام يتم وضعها في موقع عشوائي يقع ضمن حدود الكائن الأصلي، والتي يمكننا تحديدها باستخدام القيم المحلية لقياس الكائن transform.localScale. بعد وضع قطعة الحطام في موقعها العشوائي نضيف إليها قوة الانفجار التي حددناها عبر المتغير explosionPower، آخذين في الحسبان مركز الجسم الأصلي كمركز للانفجار، مما يؤدي لتناثر كافة قطع الحطام بعيدا عن مركز الجسم. بهذا تكون قطعة الحطام جاهزة، وكل ما علينا هو استدعاء التدمير المتأخر لها في حال كانت قيمة pieceLifeTime أكبر من الصفر. بعد تجهيز كافّة قطع الحطام يتم إرسال الرسالة OnBreak ومن بعدها يتم أخيرا تدمير الكائن الأصلي.

استدعاء الدّالة ()Destroy لا يعني أن Unity سيقوم بتدمير الكائن لحظيا، إنما يقوم بتأخير تدمير هذا الكائن حتى نهاية تصيير الإطار الحالي مما يعطي الفرصة للكائنات الأخرى والبريمجات الأخرى أن تستجيب وتتعامل مع نتائج تدمير الكائن خلال الإطار الحالي

بقي علينا خطوة أخيرة حتى نجعل النوافذ التي أضفناها للمبنى الموجود في المشهد scene19 قابلة للكسر. نحتاج لبريمج يعمل على ربط البريمجين Destructible و Breakable ببعضهما. هذا البريمج سيعمل على استقبال الرسالة OnDestruction وإرسال الرسالة Break بناء على ذلك. البريمج بسيط التركيب ويسمى BreakOnDestruct وهو موضح في السرد 60. النتيجة النهائية يمكن أيضا مشاهدتها وتجربتها في المشهد scene19 في المشروع المرفق.

```
using UnityEngine;
using System.Collections;

public class BreakOnDestruct : MonoBehaviour {

	void Start () {

	}

	void Update () {

	}

	//قم باستقبال الرسالة OnDestruction
	//ومن ثم إرسال الرسالة Break
	void OnDestruction(){
		SendMessage("Break", SendMessageOptions.DontRequireReceiver);
	}
}
```

السرد 60: بريمج يستقبل الرسالة OnDestruction ويقوم عندها بإرسال الرسالة Break

تمارين

1. قم بجعل السيارة التي صنعناها في الفصل الثاني ثنائية الدفع بدلا من كونها رباعية الدفع، بحيث تندفع بقوة العجلات الخلفية فقط.

2. قم بصناعة صندوقين بخواص فيزيائية وقم بتحميلهما على السيارة التي صنعناها في الفصل الثاني ثم جرب قيادة السيارة. ما هي التغييرات التي يجب أن تجريها على كائن السيارة حتى تتمكن من حمل هذه الصناديق؟

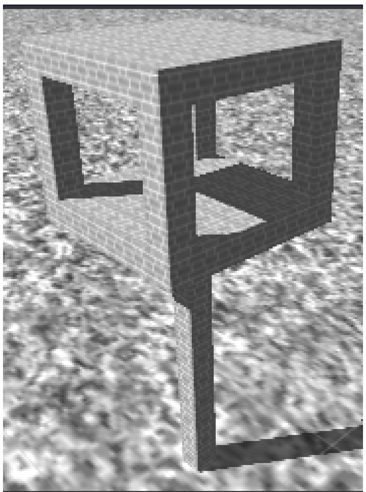
3. قم ببناء نظام تغيير سرعات للسيارة بحيث يمكن اللاعب من الانتقال بين 4 غيارات مختلفة

مستخدما المفتاح A للانتقال لغيار أعلى والمفتاح Z للانتقال لغيار أقل. تذكر أنّ العزم الأقصى للمحرك يقل كلما كان الغيار أعلى.

4. قم بكتابة بريمج خاص ببندقة خرطوش مستخدما نسخة معدلة من RaycastShooter. يجب أن يكون هذا البريمج قابلا للاستخدام مع GunInput كما يجب أن يطلق 5 إشعاعات في اتجاهات مختلفة ضمن مدى عدم الدقة في كل عملية إطلاق. أخيرا قم باستخدام هذا البريمج مع GunInput وتأكد من تعطيل إطلاق النار المستمر.

5. قم بكتابة بريمج يعمل على تدمير أي كائن بمجرد تلقيه قوة تدمير تزيد عن 50. قم بإضافة هذا البريمج للمشهد scene17 وتجربته.

6. قم ببناء برج كما في الشكل 75 ومن ثم قم بتعديل اتجاهات المسح الخاصة بالوحدات البنائية لربطها ببعضها البعض بعلاقات منطقية. على سبيل المثال يجب أن يهدم البرج كاملا في حال تهدم العمود الحامل له.



الشكل 75: البرج الخاص بالتمرين رقم 6

7. قم ببناء صندوق خشبي قابل للكسر وينكسر عند استقبال الرسالة OnRaycastHit من البريمج RaycastShooter (السرد 49 في الصفحة 145). ستحتاج بطبيعة الحال لاستخدام البريمج Breakable (السرد 59 في الصفحة 170) إضافة إلى قطع حطام بحجم وعدد مناسبين. كاختبار إضافي حاول جعل واحدة من قطع الحطام دائمة الوجود لتمثل ما يمكن اعتباره شيئا ذا قيمة للاعب.

الوحدة الخامسة: منطق اللعبة المتقدم

تعلمنا في الوحدة الثالثة مجموعة من الميكانيكيات الأساسية التي تحتاجها كثير من الألعاب، وسنتابع في هذه الوحدة الحديث عن هذه الميكانيكيات. قمت بتأخير نقاش هذه الميكانيكيات إلى ما بعد الحديث عن محاكاة الفيزياء واكتشاف التصادمات، وذلك لأن معظمها تعتمد على التصادمات. فلا قيمة مثلا للحديث عن الأبواب والأقفال إن لم نكن قادرين على اكتشاف التصادم بين هذه الأبواب وكائن اللاعب وبالتالي منعه من المرور عبرها وهي مغلقة.

بعد الانتهاء من هذه الوحدة يتوقع منك:

- صناعة أبواب وأقفال ومفاتيح
- برمجة ألغاز بسيطة وتراكيب لفك الأقفال
- برمجة صحة اللاعب وفرصه ونقاطه
- برمجة عدد من الأسلحة مع إمكانية إعادة التعبئة وتحديد الذخيرة

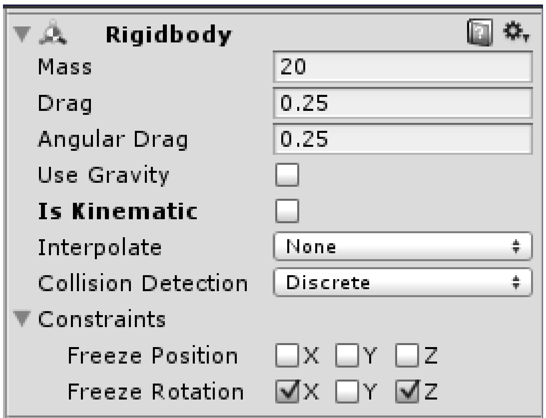
الفصل الأول: الأبواب والأقفال والمفاتيح

سنتناول في هذا الفصل نوعين من الأبواب: الأبواب الدوّارة والأبواب المنزلقة. النوع الأول أقصد به الأبواب الشائعة التي نراها في معظم الأماكن، وهي تدور حول محورها العمودي الموجود في أقصى يمين أو يسار الباب. النوع الآخر – الأبواب المنزلقة – أعني به الأبواب المتحركة التي تفتح وتغلق بالحركة على بعد واحد كما في أبواب معظم المصاعد.

لتكن البداية مع الأبواب الدوّارة والتي سنقوم ببنائها مستخدمين مكوّنا فيزيائيا جديدا يسمى Hinge Joint وتعني بالعربية "المفصل الرزي". هذا النوع من المفاصل يسمح بحركة دورانية على مستوى واحد فقط فتحا وإغلاقا، تماما كما في مفصل ركبة الإنسان. نظرا لطبيعة حركة هذا المفصل يتضح لنا أنه مناسب ليستخدم مع الأبواب الدوّارة التي نرغب بعملها. سنبدأ أولا بعمل غرفة كبيرة نسبيا مكونة من أرضية وأربع جدران، ومن ثم نفصلها لقسمين بجدارين آخرين بنهما فتحة صغيرة بحجم الباب الذي سنضيفه. نقوم بعدها بإضافة كائن الباب كما في الشكل 76 ومن ثم نضيف له مكوّن الجسم الصلب ونقوم بضبطه كما في الشكل 77. من المهم هنا ملاحظة زيادة قيمة angular drag من أجل جعل الحركة الدورانية للباب معقولة، عدا عن ذلك سيكون الباب خفيفا جدا.



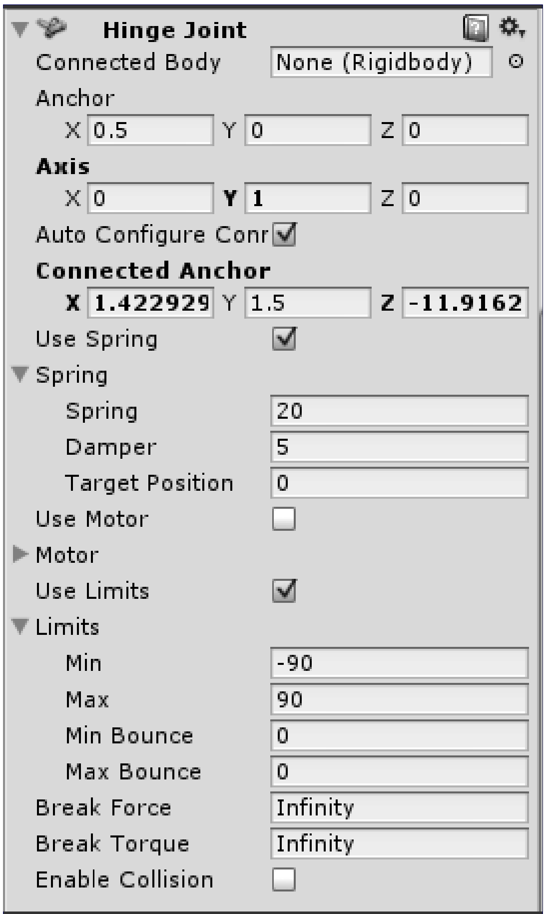
الشكل 76: الباب الدوّار



الشكل 77: ضبط الجسم الصلب الخاص بالباب الدوّار

علينا الآن أن نقوم بإضافة مكوّن المفصل الرزي إلى كائن الباب وضبطه كما في الشكل 78. هذا المفصل هو عبارة عن مكوّن فيزيائي يتأثر بالقوى الفيزيائية المختلفة؛ لذا فإنّنا لن نحتاج لأي بريمج أو ضغط أي مفتاح لفتح هذا الباب، بل كا ما علينا هو تطبيق قوة دفع بمقدار كافٍ عليه حتى يفتح.

يمكن إضافة مكوّن المفصل الرزي عن طريق القائمة Component > Physics > Hinge Joint



الشكل 78: مكوّن المفصل الرزي والقيم اللازمة لعمل باب دوّار باستخدامه

أول ما يمكن ملاحظته هنا هو أننا أمام مكوّن كبير الحجم نسبيا وفيه عدد لا بأس به من المتغيرات، إلاّ أننا سنتعامل مع بعضها فقط حتى نصل للسلوك المطلوب. البداية مع المتغيرين Anchor و Axis واللذان يحددان موقع واتجاه محور الدوران الخاص بهذا المفصل. بما أننا نقوم بعمل باب دوّار، علينا أن نضع المحور في أقصي يمين أو أقصى يسار الباب، كما يجب أن يكون اتجاه المحور عموديا أي باتجاه المحور y. بما أن قيمة الحجم z تمثل سماكة الباب وقيمة الحجم x تمثل عرضه، بالتالي فإنّ موقع المحور (أي المتغير Anchor) يجب أن يكون على أحد طرفي المحور المحلي x الخاص بكائن الباب وهو هنا (0.5, 0, 0) أي أقصى يمين الباب. كذلك الأمر بالنسبة لاتجاه المحور Axis والذي يجب أن يكون نحو الأعلى أي (0, 1, 0). التغيير الآخر الذي سنقيم بإجرائه هو تفعيل الخيار Use Spring والذي يعمل على تطبيق قوة دوران تعيد الباب إلى وضعه الأصلي حين لا يكون هناك أي قوى خارجية أكبر تؤثر عليه. بناء على هذا الخيار يجب أن يتم ضبط قيم كل من Spring و Damper بشكل مناسب بحيث لا تكون قوية جدا أو ضعيفة جدا. القيم الموضحة في الشكل 78 تم ضبطها لتتناسب مع الشخصية الفيزيائية التي قمنا بعملها في الفصل الثالث من الوحدة الرابعة. أخيرا علينا أن نقوم بتفعيل الخيار Use Limits والذي يسمح لنا بتحديد قيم دوران قصوى للباب في كلا الاتجاهين. في هذه الحالة قمنا بضبط قيم

المتغيرين Min و Max على 90- و 90 بحيث نسمح بدفع الباب من كلا جانبيه ونسمح بدورانه 90 درجة كحد أقصى. يمكنك الآن أن تضيف شخصية فيزيائية للمشهد وتقوم بتجربة الباب، حيث يمكن فتحه بمجرد الاندفاع عبره.

سنقوم الآن بقفل هذا الباب ونطلب من اللاعب أن يمتلك مفتاحا ليتمكن من فك قفله والمرور عبره. المفتاح بطبيعة الحال سيكون كائنا قابلا للجمع وتتم إضافته لمحتويات حقيبة اللاعب حين جمعه. للتذكير فقد قمنا سابقا بنقاش موضوع الأشياء القابلة للجمع وتحديدا البريمج Collectable (السرد 26 في الصفحة 85) إضافة إلى حقيبة اللاعب والبريمج InventoryBox (السرد 29 في الصفحة 89). كل ما يلزمنا هو تعديل بسيط على البريمج InventoryBox بحيث نضيف له مساحة لتخزين المفاتيح التي يقوم اللاعب بجمعها. النسخة المعدلة من البريمج موضحة في السرد 61.

```
using UnityEngine;
using System.Collections.Generic;

public class InventoryBox : MonoBehaviour {

	//المبلغ المالي الذي يمتلكه اللاعب
	public int money = 0;

	//ما هي المفاتيح التي يمتلكها اللاعب حاليا؟
	public List<string> keys;

	void Start () {

	}

	void Update () {

	}
}
```

السرد 61: النسخة المعدّلة من البريمج InventoryBox

الآلية التي سنستخدمها لعمل الأقفال تعتمد على وجود كلمة سرية خاصّة بكل مفتاح. هذه الكلمة تكون مخزنة على شكل نص في كائن المفتاح، ويمكن استخدامها لفتح كافّة الأبواب المقفلة بنفس الكلمة. القائمة keys الموجودة في البريمج InventoryBox تحتوي على قائمة بكافّة الكلمات السرية للمفاتيح التي يحملها اللاعب حاليا. ما يتوجب علينا عمله الآن هو أولا: إضافة مفتاح قابل للجمع يقوم بإعطاء اللاعب الكلمة السرية للمفتاح بمجرد جمعه، و ثانيا: عمل قفل لا يفتح إلاّ إذا قام اللاعب بتزويد الكلمة السرية المطابقة لتلك المستخدمة في القفل. من المهم هنا الإشارة إلى أنّ آلية الكلمات السرية هذه داخلية وغير مرئية للاعب، فلن نطلب منه حفظ أو كتابة أي كلمات، إنّما نقوم بذلك تلقائيا عند جمع المفاتيح وفك الأقفال. أي أنّ مهمّة اللاعب تتلخص في إيجاد المفتاح المناسب للباب.

لنبدأ بالمفتاح القابل للجمع، والذي سيعتمد على آلية معدّلة لما قمنا به في الفصل الثاني من الوحدة

الثالثة، وسيعتمد على اكتشاف التصادمات بدلا من المرور على كافّة الأشياء القابلة للجمع داخل المشهد. البداية مع البريمج CollectableKey والذي يستجيب للرسالّة Collect عن طريق تزويد اللاعب بالكلمة السرية المخزنة في المفتاح. هذا البريمج موضح في السرد 62.

```
using UnityEngine;
using System.Collections;

public class CollectableKey : MonoBehaviour {

	//الكلمة السرية الخاصّة بهذا المفتاح
	public string key;

	void Start () {

	}

	void Update () {

	}

	//Collect استقبال الرسالة
	public void Collect(GameObject owner){
		//قم بإيجاد بريمج حقيبة اللاعب الخاص بالمالك
		//ومن ثم قم بإضافة الكلمة السرية المخزنة في المفتاح
		//إلى قائمة الكلمات السرية الموجودة في الحقيبة
		InventoryBox box = owner.GetComponent<InventoryBox>();
		if(box != null){
			box.keys.Add(key);
			//قم أخيرا بتدمير كائن المفتاح
			Destroy (gameObject);
		}
	}
}
```

السرد 62: البريمج الخاص بالمفتاح القابل للجمع والذي يعطي اللاعب الكلمة السرية للمفتاح بمجرد جمعه

ما يقوم به هذا البريمج هو التأكد من كون المجمّع owner يحتوي على البريمج Inventory Box أي يمتلك حقيبة تمكنه من جمع المفاتيح. عن التأكد من وجودها يقوم بإضافة الكلمة السرية للمفتاح والموجودة في المتغير key إلى القائمة box.keys ومن ثم يقوم أخيرا بتدمير كائن المفتاح وحذفه من المشهد. عملية الحذف مهمة طبعا لإعطاء اللاعب انطباعا بأنه أخذ المفتاح فعلا. لو تساءلنا الآن: من أين يمكن للاعب أن يحصل على المفتاح؟ الجواب قد يكون بأكثر من طريقة: يمكن أن يتناوله من مكان ما على الأرض مثلا، كما يمكن أن يُعطى له من قبل شخصية أخرى في اللعبة وهلم جرا. المهم هو أن تنتهي العملية بإرسال الرسالة Collect إلى كان المفتاح مصحوبة بمرجع للاعب عبر المتغير owner. الطريقة التي سنستخدمها هي الجمع عن طريق اللمس، لكننا هذه المرة سنستفيد من المحاكي الفيزيائي وقدرته على اكتشاف التصادمات، بحيث نقوم بالجمع عند التصادم بين اللاعب وبين الكائن القابل للجمع. هذه العملية يقوم بها البريمج CollisionCollector والموضح في السرد 63.

```
using UnityEngine;
using System.Collections;

public class CollisionCollector : MonoBehaviour {

	void Start () {

	}

	void Update () {

	}

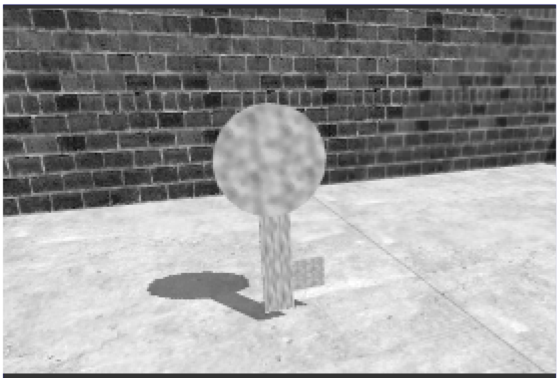
	//قم بإرسال رسالة الجمع عند اكتشاف التصادم
	void OnCollisionEnter(Collision col){
		SendCollectMessage(col.gameObject);
	}

	//قم بإرسال رسالة الجمع عند التصادم مع محفّز
	void OnTriggerEnter(Collider col){
		SendCollectMessage(col.gameObject);
	}

	void SendCollectMessage(GameObject target){
		//قم بإرسال رسالة الجمع Collect إلى الجسم الذي تم التصادم معه
		//قم بتزويد الكائن نفسه كمالك لما سيتم جمعه
		target.gameObject.SendMessage("Collect",
			gameObject, //owner تذهب هذه القيمة للمتغير
			SendMessageOptions.DontRequireReceiver);
	}
}
```

السرد 63: بريمج جمع الأشياء بناء على اكتشاف التصادم بينها وبين كائن اللاعب

يتعامل هذا البريمج مع النوعين المحتملين من التصادمات وهي التصادمات مع الأجسام الصلبة عن طريق ()OnCollisionEnter والتصادمات مع المحفّزات والتي تتعامل معها الدّالة ()OnTriggerEnter. بناء على التصادم المُكتشف يتم إرسال الرسالة Collect إلى الكائن الذي تم التصادم معه وتزويد الكائن الذي يحمل هذا البريمج (وهو هنا كائن اللاعب) عبر المتغير owner ليكون هو من يحصل على أي شيء قابل للجمع إن وُجد. بهذه الطريقة يمكننا ضمان جمع أي كائن يحتوي على البريمج Collectable طالما احتوى هذا الكائن على مكوّن تصادم Collider. لنلخص الآن ما الذي علينا فعله: نحتاج لكائن يمثل شخصية فيزيائية للاعب ويحتوي على الكاميرا مضافة كابن له. يجب أن يحتوي هذا الكائن على البريمجات PhysicsCharacter و FPSInput و inventoryBox و CollisionCollector. إضافة لذلك علينا أن نضيف كائنا يمثل المفتاح الذي سيتم التقاطه ونضيف إليه البريمج CollectableKey. يمكنك مثلا عمل مفتاح بسيط مستخدما الأشكال الأساسية كما في الشكل 79.



الشكل 79: شكل مفتاح بسيط تم إنشاؤه باستخدام الأشكال الأساسية

بعد إضافة البريمج CollectableKey لهذا الكائن علينا أن نقوم باختيار الكلمة السرية التي تحدد ما هي الأقفال التي يمكن لهذا المفتاح أن يفتحها ومن ثم نقوم بكتابة هذه الكلمة في الخانة key عن طريق نافذة الخصائص. لنستخدم مثلا الكلمة "door1" لتمييز هذا المفتاح، بالتالي عندما يقوم اللاعب بالتقاطه فإن الكلمة door1 ستضاف إلى القائمة keys الموجودة في البريمج InventoryBox والذي يمثل حقيبة اللاعب. كل ما تبقى علينا الآن هو كتابة بريمج القفل وإضافة لكان الباب الدوّار الموجود لدينا. بما أننا قمنا باستخدام مكوّن المفصل الرزي لعمل الباب وهو بطبيعة الحال مكوّن يقع تحت تحكم المحاكي الفيزيائي، فإن عملية قفل هذا الباب ستكون ببساطة عبارة عن تجميد موقعه ودورانه من خلال مكوّن الجسم الصلب، بالتالي لن يستجيب الباب لأي قوى خارجية تؤثر عليه ولن يتحرك من مكانه. البريمج PhysicsDoorLock الموضح في السرد 64 يقوم بمهمة قفل الأبواب ذات الخصائص الفيزيائية.

```
using UnityEngine;
using System.Collections.Generic;

public class PhysicsKeyLock : MonoBehaviour {

        //الكلمة السرية اللازمة لفك هذا القفل
        public string unlockKey;

        void Start () {
                Lock ();
        }

        void Update () {

        }

        //true إلى rigidbody.isKinematic قم بقفل الباب عن طريق تغيير قيمة
        public void Lock(){
                rigidbody.isKinematic = true;
        }

```

```
	//قم بمحاولة فك القفل مستخدما قائمة من الكلمات السرية
	public void Unlock(ICollection<string> keys){

		if(!rigidbody.isKinematic){
			return;
		}

		//إذا تطابقت إحدى الكلمات مع كلمة فك القفل يتم فكه تلقائيا
		foreach(string key in keys){
			if(unlockKey.Equals(key)){
				//قم بإبلاغ البريمجات الأخرى بنجاح عملية فك القفل
				SendMessage("OnUnlock",
					SendMessageOptions.DontRequireReceiver);

				rigidbody.isKinematic = false;
				return;
			}
		}

		//قم بإبلاغ البريمجات الأخرى بفشل عملية فك القفل
		SendMessage("OnUnlockFail",
			SendMessageOptions.DontRequireReceiver);
	}
}
```

السرد 64: بريمج يعمل على قفل الأبواب الفيزيائية مستخدما كلمة سرية للقفل

بتغييرنا للقيمة rigidbody.isKeinematic إلى true، فإننا نخبر المحاكي الفيزيائي أن القوى الخارجية غير مسموح لها التأثير على موقع ودوران الباب، إلا أن الأجسام الأخرى تستمر في التصادم معه ويمكنه إيقاف حركتها. يبدأ البريمج باستدعاء الدّالة ()Lock والتي تقوم بتغيير قيمة rigidbody.isKinematic إلى true. عندما يحاول أي كائن آخر أو بريمج آخر فك القفل فإنّه يتوجب عليه تزويد الدّالة ()Unlock بقائمة من الكلمات السرية (أي المفاتيح)، فإذا تطابق أحد عناصر هذه القائمة مع الكلمة المخزنة في المتغير unlockKey يتم فتح القفل. لاحظ أننا قمنا باستخدام نوع المتغير ICollection للتعبير عن القائمة، وهو نوع عام يشمل كثيرا من أنواع المجموعات المختلفة. نتيجة لذلك، فإنّ الدّالة ()Unlock قادرة على استقبال قائمة من الكلمات <List<string أو مصفوفة []string مما يعطينا مرونة أكبر في التعامل معها. قبل فحص قائمة المفاتيح يتم التأكد من أن القفل مغلق، وذلك عن طريق فحص قيمة rigidbody.isKinematic. فإذا كانت قيمته false دل ذلك على أن القفل مفتوح أصلا ولا معنى لمحاولة فتحه مرة أخرى، بالتالي يتوقف تنفيذ الدّالة. إمّا عدا ذلك فإن القفل يكون مغلقا بالتالي يمكننا أن نحاول فتحه مستخدمين عناصر القائمة المزودة لنا عبر المتغير keys. كل ما نفعله هو المرور على عناصرها واحدا تلو الآخر ومقارنة كل عنصر بقيمة المتغير unlockKey، فإن وجدنا تطابقا بين أحد المفاتيح مع هذه القيمة نقوم بفتح القفل عن طريق تغيير قيمة rigidbody.isKinematic إلى true ومن ثم إعلام باقي البريمجات بأنّه تم فك القفل عن طريق إرسال الرسالة OnUnlock. من ناحية أخرى، إذا تم المرور على كافّة العناصر دون إيجاد تطابق بين أي منها وقيمة unlockKey فإنّ عملية فتح القفل تفشل ويبقى على حاله، كما يتم إرسال الرسالة OnUnlockFail لباقي البريمجات لإعلامها بفشل عملية فتح القفل.

بعد الانتهاء من كتابة هذا البريمج يتوجب علينا أن نضيفه إلى كائن الباب الذي سبق وأنشأناه، ومن ثم نضبط قيمة unlockKey على door1 وهي نفس القيمة التي استخدمناها للمفتاح. بذلك نكون قد ربطنا المفتاح بالباب بحيث يمكنه فك قفله إن امتلكه اللاعب. بقي علينا أن نحدد متى تتم محاولة فك القفل، بمعنى آخر من سيقوم باستدعاء الدّالة ()Unlock ومتى سيقوم باستدعائها. الخيار الأوضح هو أن اللاعب يستدعي هذه الدّالة بمجرد ملامسته للباب، ويقوم بتزويدها بما يمتلكه من مفاتيح في حقيبته InventoryBox.keys لعل أحد هذه المفاتيح يتطابق مع قفل الباب. البريمج TouchUnlocker الموضح في السرد 65 يقوم بهذه المهمة.

```
using UnityEngine;
using System.Collections;

public class TouchUnlocker : MonoBehaviour {

	void Start () {

	}

	void Update () {

	}

	//قم بإرسال الرسالة Unlock إلى الجسم الذي يتم التصادم معه
	void OnCollisionEnter(Collision col){
		//استخرج مرجعا لبريمج حقيبة اللاعب
		InventoryBox box = GetComponent<InventoryBox>();

		//قم بتجربة كافّة المفاتيح الموجودة في الحقيبة
		col.gameObject.SendMessage("Unlock",
			box.keys, //مجموعة المفاتيح المستخدمة في محاولة فك القفل
			SendMessageOptions.DontRequireReceiver);

	}

}
```

السرد 65: البريمج الخاص بمحاولة فك قفل الباب بمجرد ملامسة اللاعب له

عندما يلمس اللاعب كائنا ما في المشهد فإنه يقوم بإرسال الرسالة Unlock لهذا الكائن متضمنة قائمة الكلمات السرية للمفاتيح التي يمتلكها. فإذا كان الكائن الذي تم التصادم معه بابا مقفلا فإنه سيستقبل الرسالة ويحاول فك القفل مستخدما قائمة الكلمات المزودة من قبل اللاعب. أما إذا لم يكن هذا الكائن بابا مقفلا فإنّ الرسالة تهمل ولا يحدث أي شيء. يمكنك تجربة البابين المقفل والمفتوح في المشهد scene20 في المشروع المرفق.

النوع الآخر من الأبواب الذي سنناقشه هو الأبواب المنزلقة، وسنحتاج لأجلها لكتابة بريمج خاص حيث لا يوجد مكوّن فيزيائي يمكن استخدامه بشكل مباشر. بشكل مبسط فإنّ الباب المنزلق هو كائن يتحرك

على محوره المحلي x فتحا وإغلاقا، لكن إنشاء باب كامل الوظائف يحتاج بطبيعة الحال لأكثر من ذلك؛ لأننا نحتاج لوظائف أخرى أساسية كالفتح والإغلاق والقفل وفك القفل. في الحالة السابقة حين تعاملنا مع الباب الدوّار قام مكوّن المفصل الرزي بهذه الوظائف نيابة عنا، أما الآن فإنّنا نحتاج لعمل هذه الوظائف بأنفسنا. من أجل ذلك نحتاج للبريمج GeneralDoor الموضح في السرد، والذي يحتوي على الوظائف الأساسية للأبواب بغض النظر عن الكيفية الفعلية لفتحها وإغلاقها.

```
using UnityEngine;
using System.Collections.Generic;

public class GeneralDoor : MonoBehaviour {

	//هل الباب مفتوح ابتداء؟
	public bool initiallyOpen = false;

	//الكلمة السرية لفك قفل الباب
	public string unlockKey;

	//متغير لتخزين حالة الباب داخليا
	bool isOpen;

	//متغير لتخزين حالة القفل داخليا
	bool locked;

	void Start () {
		//قم بقفل الباب إن كان هناك كلمة سرية لقفله
		locked = !string.IsNullOrEmpty(unlockKey);
		//قم بضبط الحالة الابتدائية للباب
		isOpen = initiallyOpen;
	}

	void Update () {

	}

	//قم بفتح الباب إن لم يكن القفل يمنع ذلك
	public void Open(){
		if(!locked){
			isOpen = true;
		}
	}

	//قم بإغلاق الباب إن لم يكن القفل يمنع ذلك
	public void Close(){
		if(!locked){
			isOpen = false;
		}
	}

	//قم بقفل الباب
	public void Lock(){
		locked = true;
	}
```

```
	//حاول فك القفل مستخدما مجموعة من الكلمات السرية
	public void Unlock(ICollection<string> keys){
		//قم بفحص حالة القفل الحالية أولا
		if(!IsLocked()){
			return;
		}
		//جرب جميع الكلمات السرية في القائمة لمحاولة فك القفل
		foreach(string key in keys){
			if(key.Equals(unlockKey)){
				//قم بإبلاغ البريمجات الأخرى بنجاح عملية فك القفل
				SendMessage("OnUnlock",
					SendMessageOptions.DontRequireReceiver);

				locked = false;
				return;
			}
		}
		//قم بإبلاغ البريمجات الأخرى بفشل محاولة فك القفل
		SendMessage("OnUnlockFail",
			SendMessageOptions.DontRequireReceiver);
	}

	//هل الباب مقفل حاليا؟
	public bool IsLocked(){
		return locked;
	}

	//هل الباب مفتوح حاليا؟
	public bool IsOpen(){
		return isOpen;
	}

	//قم بتبديل حالة الباب بين الفتح والإغلاق
	public void Switch(){
		if(IsOpen()){
			Close();
		} else {
			Open();
		}
	}
}
```

السرد 66: البريمج الخاص بالوظائف الأساسية للأبواب

لعلك لاحظت أن جميع الدّوال في هذا البريمج تتعامل مباشرة مع الحالة الداخلية للباب ويمكن من خلالها تغيير هذه الحالة من حيث الفتح والإغلاق أو القفل وفك القفل. جميع هذه الدّوال تسمح لك بتغيير الحالة الداخلية للباب بشكل مباشر بمجرد استدعائها، مع مراعاة وجوب تزويد الكلمة السرية الصحيحة لفك القفل في حالة الدّالة ()Unlock. ففي هذه الحالة ستبقى قيمة المتغير locked على حالها إن كانت true إذا لم يتم تزويد الدّالة بالكلمة السرية الصحيحة لفك القفل. السؤال الآن هو كيف نستفيد من هذا البريمج الذي يمثل حالة الباب في عمل باب منزلق؟ البريمج SlidingDoor يجيب على

هذا السؤال حيث يقوم بشكل مستمر بفحص حالة الباب عن طريق استدعاء ()IsOpen وتحريك الباب فعليا بناء على قيمتها. هذا البريمج موضح في السرد 67.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(GeneralDoor))]
public class SlidingDoor : MonoBehaviour {

      //الموقع النسبي للباب عند الفتح
      public Vector3 slidingDirection = Vector3.up;
      //سرعة حركة الباب عند الفتح والإغلاق
      public float speed = 2;
      //متغيرات داخلية لتخزين مواقع الفتح والإغلاق
      Vector3 originalPosition, slidingPosition;
      //GeneralDoor مرجع لبريمج الباب
      GeneralDoor door;
      //تخزين حالة الباب المنزلق
      SlidingDoorState state;

      void Start () {
            //تعيين القيم الأولية للمتغيرات
            door = GetComponent<GeneralDoor>();
            originalPosition = transform.position;
            slidingPosition = transform.position + slidingDirection;
            state = SlidingDoorState.close;
      }

      void Update () {
            if(door.IsOpen()){
                  //يجب أن يكون الباب مفتوحا
                  if(state != SlidingDoorState.open){
                        //الباب ليس مفتوحا مما يعني أنه يجب أن نحركه
                        //بشكل سلس باتجاه موقع الفتح
                        transform.position =
                                    Vector3.Lerp(
                                          transform.position,
                                          slidingPosition,
                                          Time.deltaTime * speed);

                        float remaining =
                              Vector3.Distance(
                                    transform.position, slidingPosition);

                        //التحقق من وصول الباب لموقع الفتح
                        if(remaining < 0.01f){
                              //Open position reached:
                              //change state of the door
                              state = SlidingDoorState.open;
                              transform.position = slidingPosition;
                              //قم بإبلاغ البريمجات الأخرى باكتمال عملية فتح الباب
                              SendMessage("OnOpenComplete",
                                 SendMessageOptions.DontRequireReceiver);
```

```

                } else if(state != SlidingDoorState.openning){
                    //الباب بدأ يفتح للتو
                    //قم بإرسال رسالة تبلغ البريمجات الأخرى بالأمر
                    SendMessage("OnOpenStart",
                      SendMessageOptions.DontRequireReceiver);

                    state = SlidingDoorState.openning;
                }
            }
        } else {
            //يجب أن يكون الباب مغلقا
            if(state != SlidingDoorState.close){
                //الباب ليس مغلقا، لذا يجب أن يتم تحريكه
                //بشكل سلس باتجاه موقع الإغلاق
                transform.position =
                    Vector3.Lerp(
                            transform.position,
                            originalPosition,
                            Time.deltaTime * speed);
                float remaining =
                    Vector3.Distance(
                        transform.position, slidingPosition);

                //التحقق من وصول الباب لموقع الإغلاق
                if(remaining < 0.01f){
                    //تم الوصول لموقع الإغلاق
                    //قم بتغيير حالة الباب
                    state = SlidingDoorState.close;
                    transform.position = originalPosition;
                    //قم بإبلاغ البريمجات الأخرى باكتمال عملية الإغلاق
                    SendMessage("OnCloseComplete",
                      SendMessageOptions.DontRequireReceiver);

                } else if(state != SlidingDoorState.closing){
                    //بدأ الباب بالإغلاق للتو
                    //أرسل رسالة تبلغ البريمجات الأخرى بهذا الأمر
                    SendMessage("OnCloseStart",
                      SendMessageOptions.DontRequireReceiver);

                    state = SlidingDoorState.closing;
                }
            }
        }
    }

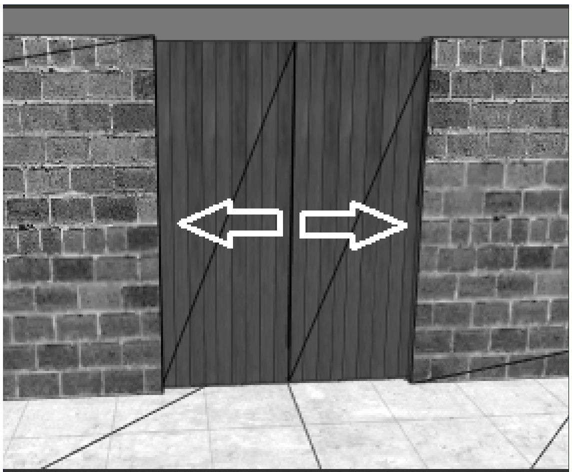
    void OnCollisionEnter(Collision col){
        if(state == SlidingDoorState.closing){
            //شيء ما أعاق إغلاق الباب
            //قم بإبلاغ البريمجات الأخرى بحدوث هذا الأمر
            SendMessage("OnCloseInterruption",
                col.gameObject,
                SendMessageOptions.DontRequireReceiver);
        }
    }
```

```
106.
107.      //مُعَدَّد خاص بالحالات المختلفة للباب المنزلق
108.      enum SlidingDoorState{
109.            open, close, openning, closing
110.      }
111. }
```

السرد 67: البريمج الخاص بالباب المنزلق

قبل الخوض في تفاصيل هذا البريمج أود التنويه إلى استخدام الحاشية RequireComponent المستخدمة قبل تعريف البريمج. هذه الحاشية تعني أن أي كائن يضاف إليه هذا البريمج يجب أن يحتوي على مكوّن محدد، وفي هذه الحالة المكوّن هو البريمج GeneralDoor. السبب في ذلك هو أن البريمج SlidingDoor يعتمد في أدائه على GeneralDoor ولا يعمل بدونه بشكل صحيح. لذا يتم فحص وجود البريمج GeneralDoor. إذا قمت بإضافة SlidingDoor إلى كائن لا يحتوي على GeneralDoor فإن Unity يقوم بإضافة هذا الأخير تلقائيا. كذلك الأمر لو حاولت حذف البريمج GeneralDoor عن كائن يحتوي على SlidingDoor سيمنع Unity ذلك حيث أنه يتوجب عليك أولا حذف كافّة البريمجات التي تحتاج له ومنها SlidingDoor.

بالعودة الآن لتفاصيل البريمج، نلاحظ وجود المتغير slidingDirection والذي يمثل متجه المسافة التي يقطعها الباب حين يتحرك من موقع الإغلاق لموقع الفتح. فإذا كانت قيمته مثلا (0, 2, 0) فإنّ الباب سيتحرك مترين نحو الأعلى عند فتحه. سرعة حركة الباب يتحكم بها المتغير speed، كما أنّ حركة الفتح والإغلاق نفسها هي عبارة عن انتقال سلس عن طريق الاستيفاء بين نقطتي الإغلاق والفتح المخزنتين في المتغيرين originalPosition و slidingPosition. المتغير الأول originalPosition يأخذ قيمته عند بداية التشغيل من الموقع الحالي للباب، وهذا يعني أنّ الباب يجب أن يوضع في المشهد في حالة الإغلاق دائما. أمّا المتغير الآخر slidingPosition فهو عبارة عن ناتج جمع الموقع الأصلي مع المتجه slidingDirection. سبق وعرفنا أن حالة الباب تتم إدارتها عن طريق البريمج GeneralDoor، إضافة لذلك نقوم بإدارة الحالات الانتقالية للباب المنزلق عن طريق المُعَدَّد SlidingDoorState والذي نحفظ فيه قيمة الحالة الحالية للباب المنزلق (مفتوح opened، مغلق closed، يفتح opening، يغلق closing) حيث يتم إسناد القيمة الأولية بناء على قيمة المتغير door.initiallyOpen. الشكل 80 يوضح بابا منزلقا ذو مصراعين يتحركان في اتجاهين متعاكسين عند الفتح والإغلاق.



الشكل 80: باب منزلق ذو مصراعين يتحرك كل منهما في اتجاه السهم حين فتحه

أثناء تنفيذ الدّالة ()Update يقوم البريمج باستدعاء ()door.IsOpen وفحص القيمة التي ترجعها. فإذا كانت القيمة true يعني أن الباب يجب أن يكون مفتوحا، وإن لم يكن كذلك يجب فتحه. لأجل ذلك نقوم بفحص الحالة الحالية للباب المنزلق state والتأكد من أنها تساوي SlidingDoorState.open. فإن لم تكن كذلك فهي واحدة من الثلاث الأخريات: إمّا أن الباب مغلق SlidingDoorState.closed أو أنّه يتم إغلاقه SlidingDoorState.closing أو أنّه يتم فتحه SlidingDoorState.opening. في الحالتين الأوليين وهما حالتا الإغلاق علينا أن نقوم بتغيير الحالة إلى الفتح SlidingDoorState.opening، أمّا إذا كانت القيمة تساوي الحالة الثالثة فإننا نقوم بتحريك الباب بشكل سلس باتجاه موقع الفتح slidingPosition. هذه الحركة السلسة تتم عن طريق الدّالة ()Vector3.Lerp والتي سبق شرح طريقة عملها. آخذين بعين الاعتبار أن المنطقة الميتة لفتح الباب أو إغلاقه هي 0.01 أي سنتيمترا واحدا، فإننا نقوم بوضع الباب مباشرة في الموقع slidingPosition إذا قلت المسافة بين موقع الباب الحالي وهذه النقطة عن مقدار المنطقة الميتة. وبعدها نقوم بتغيير حالة الباب إلى الحالة الجديدة وهي "مفتوح"
SlidingDoorState.open. وإرسال رسالة تعلم البريمجات الأخرى بالحالة الجديدة للباب. الأمر نفسه يتم ولكن في الاتجاه المعاكس إذا كانت القيمة المُرجعة من ()door.IsOpen هي false. حيث نقوم بفحص ما إذا كانت حالة الباب هي الإغلاق SlidingDoorState.closed وتغييرها للحالة المناسبة إذا كانت غير ذلك. أخيرا يمكننا اكتشاف تصادم الباب مع أي كائن آخر أثناء الإغلاق، وفي هذه الحالة نقوم بإرسال الرسالة OnCloseInterruption مزودين معها مرجعا للكائن الذي يعيق إغلاق الباب. هذه الرسالة يمكن التعامل معها مثلا عن طريق تدمير الكائن المعيق مما يجعل الباب سلاحا يمكن للاعب استخدامه ضد الأعداء، خاصّة إن كان يمكنه التحكم بالباب من مكان بعيد.

الفصل الثاني: الألغاز وتراكيب فك الأقفال

هذا الفصل هو امتداد للفصل السابق حيث سنواصل العمل على الباب المنزلق الذي سبق وقمنا

بعمله. سنقوم بإغلاق هذا الباب مستخدمين قفلا مركزيا نفترض أنه يعمل بآلة كهربائية معينة، وعلى اللاعب أن يقوم بحل لغز بسيط حتى يتمكن من فك القفل. ما نحتاج لعمله الآن هو إضافة البريمج SlidingDoor لكل طرف من أطراف الباب الموضح في الشكل الشكل 80 ونقوم بضبط اتجاه الانزلاق عبر المتغير slidingDirection بحيث تكون مثلا (1.2, 0, 0) للشق الأيمن و (-1.2, 0, 0) للشق الأيسر من الباب. عند أضافة البريمج SldingDoor سيقوم Unity تلقائيا بإضافة البريمج GeneralDoor ذلك أن الأول يعتمد على الثاني كما وضحنا في الفصل السابق. البريمج GeneralDoor يفترض أن الباب لا يحوي قفلا طالما أن قيمة المتغير unlockKey خالية، لذا علينا أن نقوم بتحديد قيمة ما ولتكن مثلا "door2" لكل من شقي الباب المنزلق. علينا الآن أن نقوم بإضافة القفل المركزي وهو عبارة عن كائن خالٍ يحتوي على البريمجات اللازمة للتحكم بالباب المنزلق فتحا وإقفالا. أول هذه البريمجات هو CentralLock والموضح في السرد 68.

```
using UnityEngine;
using System.Collections;

public class CentralLock : MonoBehaviour {

	//كائنات الأبواب التي سيتحكم بها هذا القفل
	public GeneralDoor[] targetDoors;

	//مجموعة الكلمات السرية التي يحتاجها القفل لفك قفل الأبواب
	public string[] keys;

	//هل سيفتح الباب تلقائيا عند فك القفل؟
	public bool autoOpen = true;

	//هل سيغلق الباب تلقائيا قبل أن يتم قفله؟
	public bool autoClose = true;

	void Start () {

	}

	void Update () {

	}

	//قم بقفل كافّة الأبواب
	public void LockAll(){
		foreach(GeneralDoor door in targetDoors){
			if(autoClose){
				door.Close();
			}
			door.Lock();
		}
	}

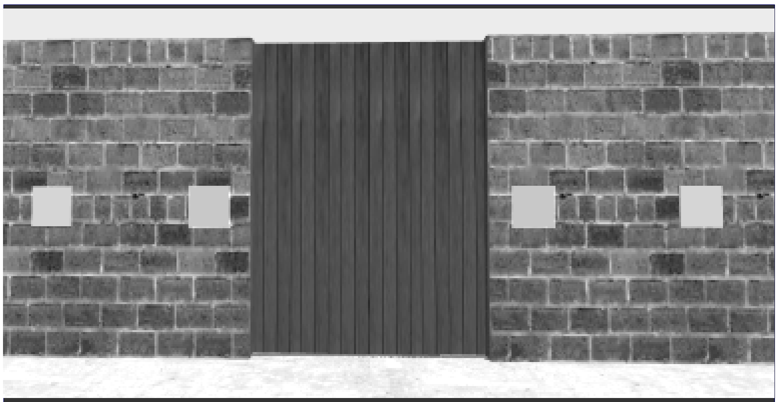
	//قم بفك قفل كافّة الأبواب مستخدما الكلمات المتوفرة
	public void UnlockAll(){
		foreach(GeneralDoor door in targetDoors){
```

```
                    door.Unlock(keys);
                    if(autoOpen){
                          door.Open();
                    }
              }
        }
}
```

السرد 68: البريمج الخاص بالقفل المركزي

يحتوى هذا البريمج على مصفوفة من الأبواب targetDoors كما يحتوي على مصفوفة أخرى من الكلمات السرية اللازمة لفتح هذه الأبواب وهي keys. عند استدعاء الدّالة ()LockAll يقوم البريمج بالمرور على جميع أبواب المصفوفة targetDoors وقفلها عن طريق استدعاء الدّالة ()door.Lock في كل مرة. إضافة إلى ذلك سيقوم البريمج بإغلاق الأبواب قبل قفلها وذلك إذا كان الخيار autoClose مفعّلا. على الجانب الآخر فإنّ الدّالة ()UnlockAll تحاول فك قفل جميع الأبواب مستخدمة مصفوفة الكلمات السرية keys مع كل باب، كما أنها تقوم بفتح الباب بعد فك قفله في حال كان الخيار autoOpen مفعّلا. باستخدامنا لمجموعة من الأبواب ومجموعة من الكلمات السرية يمكننا استخدام هذا القفل المركزي للتحكم بمجموعة أبواب لا تمتلك بالضرورة نفس الكلمة السرية لفك قفلها. هذا مفيد لو تخيلت مثلا أنك تريد عمل غرفت تحكم ذات مدخل سري يمكن للاعب من داخلها فك قفل جميع الأبواب في المرحلة الحالية بضغطة زر، وإلا فإن عليه أن يقوم بالبحث عن مفتاح لكل باب لفتحه بشكل مستقل. لاحظ أيضا أن المصفوفة targetDoors هي من نوع GeneralDoor، مما يجعلها قابلة لاستيعاب أي نوع من الأبواب قد يلزمك إضافته، وليس فقط الأبواب المنزلقة.

بعد كتابة البريمج يمكننا أن نضيفه للكائن الفارغ الذي سنستخدمه كقفل مركزي، كما يتوجب علينا أن نضيف شقي الباب الأيمن والأيسر إلى المصفوفة targetDoors عبر نافذة الخصائص حتى يصبح هذان الشقان تحت تصرف القفل المركزي. بعدها علينا أن نضيف الكلمة السرية التي استخدمناها لقفل الباب المنزلق وهي door2 إلى المصفوفة keys حتى يكون القفل المركزي قادرا على فك قفل الباب حين يُطلب منه ذلك. سنبقي على الخيارين autoOpen و autoClose مفعّلين مما يوفر علينا إضافة آلية أخرى لفتح الباب المنزلق وإغلاقه بالتالي نكتفي بالقفل المركزي ليقوم بالمهمّة. لو تأملنا الوضع الحالي للمشهد، سنجد أننا أمام باب منزلق ذو شقين أيمن وأيسر، وهذا الباب مقفل باستخدام كلمة سرية وشقاه متصلان بقفل مركزي يحمل هذه الكلمة السرية ويمكنه فك القفل وفتح الباب بمجرد استدعاء الدّالة ()UnlockAll. إذن نحن أمام سؤال واحد أخير: متى سيتم استدعاء ()UnlockAll ومن سيقوم باستدعائها؟ الجواب هو نظام الألغاز الذي سنقوم ببنائه بعد قليل. قبل الخوض في التفاصيل البرمجية لهذا النظام لنتعرف على آلية عمله: سيحتوي هذا اللغز على أربعة أزرار، وسيكون لكل زر حالتان: أخضر وأحمر، وسيحتاج اللاعب لفك القفل إلى إيجاد التركيبة المناسبة بين ألوان الأزرار الأربعة، أي أنه أمام 16 احتمالا مختلفا (أي 2^4). سنقوم بترتيب هذه الأزرار الأربعة (وهي عبارة عن مكعبات صغيرة الحجم نسبيا) حول الباب كما في الشكل 81.



الشكل 81: أزرار لغز فك القفل الأربعة موزعة حول الباب المنزلق

ما ينبغي علينا فعله الآن هو إعطاء اللاعب القدرة على التحكم بكل واحد من هذه الأزرار الأربعة وتغيير لونه من الأحمر للأخضر أو العكس. لأجل ذلك سنعيد استخدام بريمجات المحفّزات التي كنا قد كتبناها سابقا وهي SwitchableTrigger (السرد 35 في الصفحة 101) و TriggerSwitcher (السرد 37 في الصفحة 104) واللذين تم إنجازهما في الفصل الثالث من الوحدة الرابعة. بمراجعة سريعة لمحتوى البريمجين نتذكر وجود الدّالة ()SwitchState في البريمج SwitchableTrigger والتي تنقل المحفّز بين عدة حالات ويمكنها إرسال رسائل مختلفة عند الانتقال من حالة لأخرى. أمّا بالنسبة للبريمج TriggerSwitcher فهو يضاف لكائن شخصية اللاعب ليسمح له بتفعيل هذه المحفّزات عن طريق استدعاء ()SwitchState من المحفّز بالاقتراب من المحفّز والضغط على مفتاح E على لوحة المفاتيح. عند تحفيز اللاعب لأي من أزرار اللغز الأربعة، هناك ثلاث خطوات ينبغي القيام بها: الخطوة الأولى هي تغيير لون الزر نفسه من الأحمر للأخضر أو العكس، والخطوة الثانية هي التواصل مع بريمج مسؤول عن إدارة حالة اللغز وإعلامه بوجود تركيبة ألوان جديدة، مما يعني أنه يجب التأكد من أن اللاعب قد توصل للتركيبة الصحيحة لحل اللغز أم لا، والخطوة الثالثة هي محاولة فك قفل الباب باستخدام التركيبة الجديدة. وبما أننا اخترنا أن يفتح الباب تلقائيا عند فك قفله، فهذا يعني أنه بمجرد توصل اللاعب للتركيبة الصحيحة لحل اللغز سيفتح الباب تلقائيا دون الحاجة للاقتراب منه ومحاولة فتحه.

لنبدأ مع الخطوة الأولى والأسهل وهي تغيير لون الزر من الأحمر للأخضر وبالعكس. هذه المهمة يقوم بها البريمج ColorCycler والموضح في السرد 69. هذا البريمج يعتمد في عمله على تغيير لون الخامة الرئيسية لمكوّن التصيير renderer الخاص بالكائن.

```
using UnityEngine;
using System.Collections;

public class ColorCycler : MonoBehaviour {

	//مصفوفة الألوان التي سيتم التبديل بينها
	public Color[] colors;

```

```
	//الموقع الخاص باللون الحالي
	public int currentColor = 0;

	void Start () {
		renderer.material.color = colors[currentColor];
	}

	void Update () {

	}

	//قم بالتبديل للون التالي في المصفوفة أو العودة للبداية حين الوصول لنهايتها
	public void CycleColor(){
		if(colors.Length > 0){
			currentColor++;

			if(currentColor == colors.Length){
				currentColor = 0;
			}

			renderer.material.color = colors[currentColor];
		}
	}
}
```

السرد 69: بريمج يقوم بتغيير لون الكائن بين عدة قيم مخزنة في مصفوفة

ما يتوجب علينا فعله الآن هو إضافة البريمج SwitchableTrigger إلى الأزرار الأربعة التي قمنا بإضافتها حول الباب (الأفضل طبعا هو عمل قالب خاص بالأزرار ونسخه أربع مرات) ومن ثم تحديد عدد الحالات التي يمكن الانتقال بينها إلى حالتين. عند الانتقال للحالة الأولى أو الثانية ستكون النتيجة دائما هي إرسال الرسالة CycleColor والتي سيقوم البريمج ColorCycler - والذي يجب أن يكون مضافا للزر أيضا ومضاف إليه اللونان الأحمر والأخضر – باستقبالها وتغيير اللون بناء عليها. الآن علينا تنفيذ الخطوة الأخيرة وهي حلقة الوصل بين الأزرار الأربعة والبريمج CentralLock والذي يمتلك إمكانية فك قفل الباب وفتحه، وهي العملية التي يجب أن تتم بناء على حالة اللغز. بما أننا نتحدث عن أربعة أزرار متفرقة ينبغي علينا أن نمتلك آلية لمعرفة ألوان هذه الأزرار ومقارنتها بحل اللغز الصحيح الذي نختاره، وبناء على التطابق بين حالة الأزرار الأربعة والحل الصحيح نقوم بإرسال الرسالة UnlockAll للبريمج CentralLock حتى يفك قفل الباب ويقوم بفتحه. هذا البريمج هو ColorCodePuzzle الموضح في السرد 70، وهو المكان الفعلي الذي يحتوي على منطق اللغز وكيفية حله. عند تطابق حل اللاعب مع الحل الصحيح سنحتاج لإرسال الرسالة UnlockAll، أما في حال عدم التطابق علينا إرسال الرسالة LockAll.

```
using UnityEngine;
using System.Collections;

public class ColorCodePuzzle : MonoBehaviour {

	//تركيبة فك القفل أو الحل الصحيح لهذا اللغز
```

```
    public Color[] unlockCode;

    //مصادر الحصول على مدخلات اللاعب لحل اللغز
    public Renderer[] colorSources;

    //الرسائل التي سيتم إرسالها عند حل اللغز حلا صحيحا
    public TriggerMessage[] matchMessages;

    //الرسائل التي سيتم إرسالها عند الحل الخاطئ للغز أي التراكيب غير المتوافقة مع الحل
    public TriggerMessage[] mismatchMessages;

    void Start () {
    }

    void Update () {
    }

    //قم بمقارنة تركيبي اللاعب والحل الصحيح
    public void CompareCodes(){
        //نفترض أن التركيبين متطابقان
        bool match = true;

        //قم بالمرور على المدخلات ومقارنتها مع الحل الصحيح
        for(int i = 0; i < colorSources.Length; i++){
            //مجرد وجود اختلاف في مدخل واحد كافٍ لنفي التطابق بين التركيبين
          if(!colorSources[i].material.color.Equals(unlockCode[i])){
                match = false;
            }
        }

        TriggerMessage[] toSend;

        //في حال التطابق بين تركيب مدخل اللاعب وتركيب الحل الصحيح
        //علينا أن نقوم بإرسال الرسائل الخاصة بالتطابق
        if(match){
            toSend = matchMessages;
        } else {
            //عدا ذلك علينا إرسال الرسائل الخاصة بعدم التطابق
            toSend = mismatchMessages;
        }

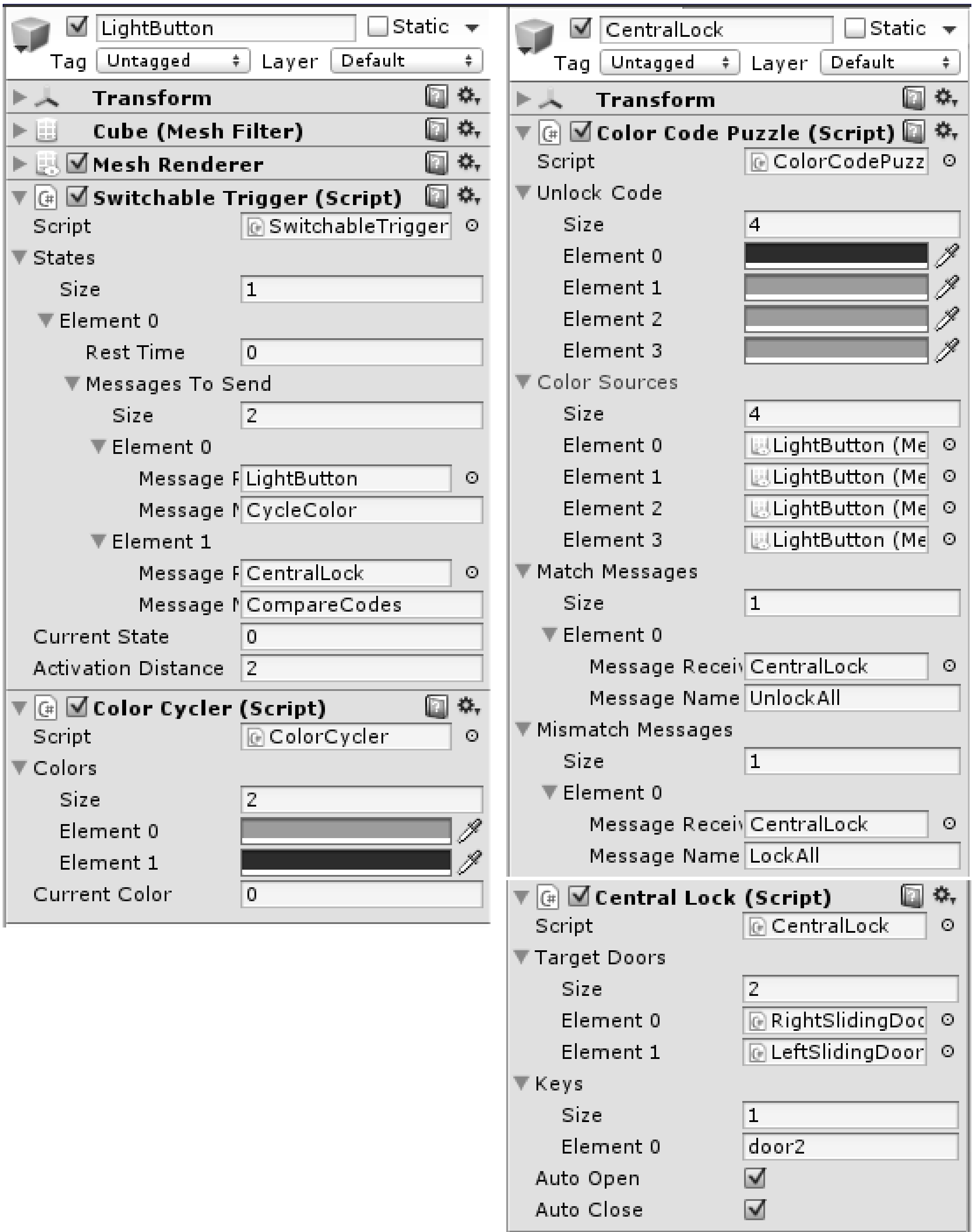
        //قم بإرسال الرسائل
        foreach(TriggerMessage msg in toSend){
            if(msg.messageReceiver != null){
                msg.messageReceiver
                            .SendMessage(
                                msg.messageName,
                             SendMessageOptions.RequireReceiver);

            }
        }
    }
}
```

السرد 70: بريمج لغز تراكيب الألوان

قمنا في هذا البريمج بإعادة استخدام TriggerMessage الذي قمنا بكتابته في الفصل الثالث من الوحدة الرابعة (السرد 35 في الصفحة 101) وقمنا بإنشاء مصفوفتين من هذا النوع. المصفوفة الأولى هي matchMessages وهي الرسائل التي سنقوم بإرسالها في حال فحص الحل وحدوث التوافق بين مُدخل اللاعب والحل الصحيح للغز والمصفوفة الثانية هي mismatchMessages وهي الرسائل التي سنرسلها في حال فحص الحل وعدم اكتشاف توافق بين مُدخل اللاعب والحل الصحيح. بدوره فإن الحل الصحيح يتم تحديده مباشرة من نافذة الخصائص وذلك عبر المصفوفة unlockCode والتي تحتوي على الترتيب الصحيح للألوان والذي سيعمل على حل اللغز وبالتالي إرسال رسالة لفك قفل الباب. بعد ذلك علينا أن نربط الأزرار الأربعة بهذا البريمج وذلك عن طريق المصفوفة colorSources والتي هي عبارة عن مصفوفة لمكونات من نوع renderer وهي التي سنعمل على استخراج الألوان منها. ما علينا فعله الآن هو إضافة كل زر من الأزرار الأربعة عبر مكوّن التصيير renderer الخاص به إلى المصفوفة colorSources وليكن ترتيبها من اليسار لليمين. لدينا الآن أربعة مصادر للألوان تأتي من مكوّنات تصيير الأزرار الأربعة إضافة لأربع ألوان تحدد الحل الصحيح موجودة في المصفوفة unlockCode. بالتالي عند تفعيل المحفّز الخاص بكل زر من الأزرار يجب أن نعاود استدعاء ()CompareCodes وذلك حتى يقوم البريمج ColorCodePuzzle بمقارنة تركيبة الألوان الجديدة وفتح الباب إن كانت صحيحة. من المهم ذكره هنا هو أن عملية مقارنة الألوان تتم برمجيا عن طريق مقارنة القيم الرقمية الخاصة بدرجات الأخضر والأحمر والأزرق والشفافية، لذا من المهم أن تكون الألوان متطابقة تماما رقميا فلن ينفع مثلا وجود درجتين مختلفتين من الأخضر بين القيم في colorSources و unlockCode لأن ذلك سيعني برمجيا أنه لا تطابق لونيا بينهما.

بقي علينا أن نذكر أن البريمج ColorCodePuzzle يجب أن تتم إضافته لنفس الكائن الذي أضفنا له البريمج CentralLock. لتلخيص كل هذه التفاصيل انظر الشكل 82 والذي يوضح مكوّنات بريمجات فك القفل عن طريق الألغاز إضافة للمتغيرات والقيم الخاصة بهذه البريمجات. يمكنك أيضا تجربة النتيجة النهائية في المشهد scene20 في المشروع المرفق.



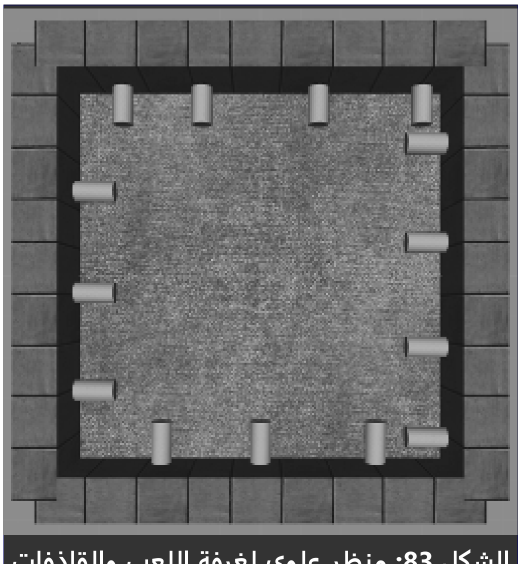
الشكل 82: ضبط قيم المتغيرات الخاصة ببريمجات أزرار لغز فك القفل إضافة للبريمجين CentralLock و ColorCodePuzzle وذلك لصناعة لغز لفك القفل عن طريق ترتيب الألوان

الفصل الثالث: صحة اللاعب والفرص والنقاط

تعتبر صحة اللاعب من الأمور الحيوية في كثير من الألعاب، حيث يبدأ اللاعب عادة بصحة مقدارها

100%، ثم تبدأ بالتغير تبعا للضربات التي يتلقاها من الخصوم أو من أي عناصر مؤذية في بيئة اللعب، هذا إضافة إلى إمكانية إعادة استرجاع الصحة الكاملة أو جزء منها عن طريق التقاط مواد علاج يمكن أن يجدها خلال اللعب. بطبيعة الحال فإن وصول صحة اللاعب للقيمة صفر تعني موته وبالتالي خسارة اللعبة، أو على الأقل خسارة فرصة واحدة ومن ثم البداية مجددا إن كان معه مزيد من الفرص. في حال وجود هذه الفرص في نظام اللعبة فإن نهاية اللعبة تكون باستنفاذ كافّة هذه الفرص. إضافة للصحة والفرص، تحتوي كثير من الألعاب على نظام نقاط محدد يجعل مقارنة الأداء بين اللاعبين وتحديد الفائزين والمتصدرين أمرا ممكنا.

سنقوم في هذا الفصل بتجميع الأفكار الثلاث، وهي الصحة والفرص والنقاط في لعبة واحدة مكتملة. فكرة اللعبة تقوم على تحكم اللاعب بمكعب محصور داخل غرفة، وعلى جدران الغرفة قاذفات تطلق طلقات باتجاهات مختلفة. الهدف من اللعبة هو النجاة من هذه الطلقات أكبر وقت ممكن قبل استنفاذ الفرص جميعها. لنفترض مثلا أننا سنعطي اللاعب 3 فرص وصحة مقدارها 100 في كل فرصة. علاوة على ذلك سيكون لدينا نوعان من الطلقات أحدهما أحمر اللون وينقص من صحة اللاعب بمقدار 10 وحدات والآخر أخضر اللون وينقص 5 وحدات. بما أنّ اللاعب الأفضل سيتمكن من الصمود فترة أطول في هذه الغرفة قبل أن يستنفذ الفرص الثلاث وتنتهي اللعبة، من المنطقي أن نقوم بحساب هذا الوقت واعتماده كنقاط لقياس أداء اللاعب. الشكل 83 يظهر شكل الغرفة التي سنقوم بعملها لهذه اللعبة، حيث قاذفات الطلقات عبارة عن أسطوانات تم تدويرها بحيث يشير سطحها العلوي (الاتجاه الموجب للمحور y) إلى داخل الغرفة.



الشكل 83: منظر علوي لغرفة اللعب والقاذفات التي تحيط بها من كل جانب

بعد ذلك علينا إزالة مكوّن التصادم من جميع القاذفات منعا للتصادم بينها وبين الطلقة المقذوفة عند إطلاقها. سنقوم أيضا بتزويد كل واحدة من هذه القاذفات بمجموعة من قوالب الطلقات المقذوفة

بحيث تختار واحدة منها عشوائيا وتطلقها في كل مرّة. أخيرا سنقوم بإضافة حد أعلى وحد أدنى لوقت الانتظار بين كل طلقتين متتاليتين. جميع المهام السابق يقوم بها البريمج PhysicsShooter الموضح في السرد 71.

```
using UnityEngine;
using System.Collections;

public class PhysicsShooter : MonoBehaviour {

	//مصفوفة تحتوي على قوالب الطلقات
	public GameObject[] projectils;

	//أقصى وأدني عدد من الثواني التي يمكن انتظارها بين الطلقتين المتتاليتين
	public float minTime = 1, maxTime = 6;

	void Start () {
		ShootRandomly();
	}

	void Update () {

	}

	void ShootRandomly(){
		//قم بإطلاق النار بعد عدد عشوائي من الثواني
		float randomTime = Random.Range(minTime, maxTime);
		Invoke("Shoot", randomTime);
	}

	void Shoot(){
		//قم باختيار طلقة عشوائية
		int index = Random.Range(0, projectils.Length);
		GameObject prefab = projectils[index];
		GameObject projectile = (GameObject)Instantiate(prefab);

		//قم بإطلاق الطلقة التي تم اختيارها
		projectile.transform.position = transform.position;
		projectile.rigidbody.AddForce
				(transform.up * 6, ForceMode.Impulse);

		//قم باستدعاء دالّة الإطلاق مرة أخرى
		ShootRandomly();
	}
}
```

السرد 71: بريمج أطلاق الطلقات العشوائية للقاذفات

كما تلاحظ فإنّ عملية الإطلاق تتم فعليا عبر إضافة قوة اندفاع للطلقة المقذوفة، وهي بدورها يتم اختيارها عشوائيا من المصفوفة projectiles. بعد كل عملية إطلاق يتم استدعاء ()ShootRandomly، وذلك حتى تقوم بتوليد رقم عشوائي لاستخدامه كوقت لتأخير عملية الإطلاق التالية. لاحظ أن القيمة العشوائية محصورة بين قيمتي المتغيرين minTime و maxTime، ويتم استخدامها مع الدّالة ()Invoke

لاستدعاء ()Shoot لاحقا. الخطوة التالية هي بناء قالبين للطلقتين الحمراء والخضراء التين تحدثنا عنهما. ما تقوم به هاتان الطلقتان هو إنقاص صحة اللاعب بمجرد لمسه، لذلك نضيف إليهما بريمجا أسميناه PainfulProjectile وهو موضح في السرد 72.

```
using UnityEngine;
using System.Collections;

public class PainfulProjectile : MonoBehaviour {

        //مقدار الصحة الذي ستنقصه الطلقة من اللاعب حين الإصابة
        public int damage;

        void Start () {

        }

        void Update () {

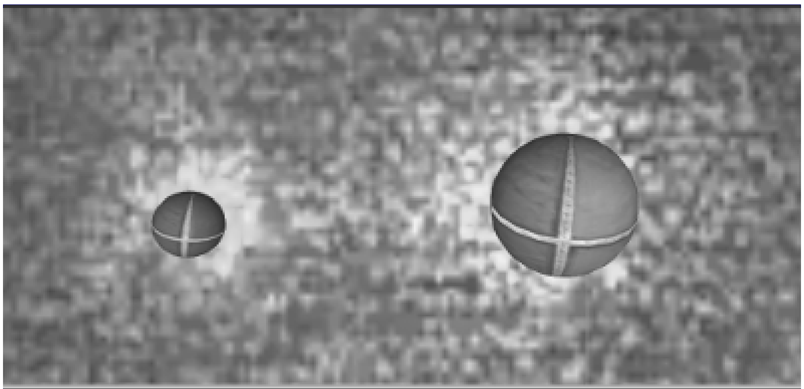
        }

        void OnCollisionEnter(Collision col){
                //قم بإرسال رسالة إنقاص الصحة إلى الجسم الذي تم الاصطدام به
                col.gameObject.SendMessage("OnPainfulHit",
                                        damage,
                                        SendMessageOptions.DontRequireReceiver);

                //قم بتدمير كائن الطلقة
                Destroy(gameObject);
        }
}
```

السرد 72: بريمج الطلقة التي تنقص من صحة اللاعب

هذا البريمج بسيط نوعا ما، فكل ما يقوم به هو إرسال رسالة إلى الجسم الذي يصطدم به ويرسل معها مقدار الإنقاص الذي يجب أن يحدث من الصحة، طبعا في حال كان الجسم المتصادم معه هو اللاعب. سنقوم بإضافة هذا البريمج لقالبين سنقوم بعملهما للطلقات مستخدمين كرتين بإكساءين أحمر وأخضر. إضافة لذلك سنضيف ضوءا نقطيا كابن لكل كرة ونقوم بتغيير لونه لتظهر الطلقة كأنها مشعّة. أخيرا علينا أن نضيف لكل قالب طلقة جسما صلبا ونقوم بتعطيل استخدام الجاذبية عن طريق إلغاء الاختيار عن Use Gravity. هذه الخطوة الأخيرة مهمة لجعل الطلقة المقذوفة تمشي بخط مستقيم دون أن تسقط على الأرض. الشكل 84 يظهر الطلقتين التين سنستخدمهما.



الشكل 84: يسارا: طلقة صغيرة تشع بلون أخضر وتنقص من صحة اللاعب بمقدار ، ويمينا: طلقة حمراء تنقص بمقدار 10

ما يتوجب عمله الآن هو إضافة قالبي هاتين الطلقتين إلى المصفوفة projectiles في قالب مدافع الإطلاق، مما يجعلهما تضافان تلقائيا إلى كافّة المدافع في المشهد. لإكمال بيئة اللعب علينا أخيرا أن نضيف اللاعب الذي سنتحكم به، وسيكون هنا عبارة عن مكعب مضاف إليه البريمج PhysicsCharacter إضافة إلى TopViewControl والموضح في السرد 73. هذا البريمج الأخير يسمح لنا بالتحكم بشخصية اللاعب من منظور علوي وتحريكه في الاتجاهات الأربع. إضافة لذلك سنمنع عملية القفز وذلك من خلال تعطيل حركة الجسم الصلب على المحور y إضافة لمنع دورانه على جميع المحاور.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(PhysicsCharacter))]
public class TopViewControl : MonoBehaviour {

      //متغير لتخزين شخصية اللاعب التي سيتم التحكم بها
      PhysicsCharacter pc;

      void Start () {
            //اعثر على بريمج شخصية اللاعب
            pc = GetComponent<PhysicsCharacter>();
      }

      void Update () {
            //تحكم بالحركة باستخدام الأسهم
            if(Input.GetKey(KeyCode.RightArrow)){
                  pc.StrafeRight();
            } else if(Input.GetKey(KeyCode.LeftArrow)){
                  pc.StrafeLeft();
            }

            if(Input.GetKey(KeyCode.UpArrow)){
                  pc.WalkForward();
            } else if(Input.GetKey(KeyCode.DownArrow)){
```

```
                    pc.WalkBackwards();
              }

        }
}
```

السرد 73: البريمج الخاص بالتحكم بشخصية اللاعب الفيزيائية من منظور علوي

من المهم أن نقوم بعمل قالب لكائن اللاعب؛ وذلك لأننا نقو بعمل لعبة يمتلك فيها اللاعب أكثر من فرصة، وبالتالي سنحتاج لتدمير وإعادة إنشاء كائن اللاعب عدة مرات. إضافة للتحكم باللاعب سنحتاج لبريمج يحدد صحة اللاعب الحالية ويتحكم بزيادتها وإنقاصها. هذا البريمج هو PlayerHealth الموضح في السرد 74، حيث يتم تمثيل صحة اللاعب على شكل عدد صحيح نقوم بزيادته أو إنقاصه مستخدمين مجموعة من الدّوال.

```
using UnityEngine;
using System.Collections;

public class PlayerHealth : MonoBehaviour {

      //القيمة الأولية لصحة اللاعب
      public int initialHealth = 100;

      //الحد الأقصى للصحة
      public int maxHealth = 100;

      //القيمة الحالية لصحة اللاعب
      int health;

      //مؤشر داخلي للاستدلال موت اللاعب
      bool dead = false;

      void Start () {
            //تأكد من أن القيمة الأولية للصحة صحيحة
            health = Mathf.Min(initialHealth, maxHealth);
            //لا يمكن أن يبدأ اللاعب اللعبة ميتا
            if(health < 0){
                  health = 1;
            }
      }

      void Update () {
            if(!dead){
                  if(health <= 0){
                        //يموت اللاعب في هذه الحالة
                        dead = true;

                        //قم بإخبار البريمجات الأخرى بموت اللاعب
                        //مع تزويد القيمة النهائية لصحته
                        SendMessage("OnPlayerDeath",
                              health,
                              SendMessageOptions.DontRequireReceiver);
```

```
			}
		}
	}

	//قم بإنقاص الصحة وإبلاغ البريمجات الأخرى بذلك
	public void DecreaseHealth(int amount){
		//لا حاجة لإنقاص الصحة إذا كان اللاعب ميتا أساسا
		if(IsDead()) return;

		health -= amount;
		SendMessage("OnHealthDecrement",
					health,
					SendMessageOptions.DontRequireReceiver);
	}

	//قم بزيادة صحة اللاعب وإعلام البريمجات الأخرى بذلك
	public void IncreaseHealth(int amount){
		//لا يمكن زيادة الصحة للاعب ميت
		if(IsDead()) return;

		//قم بزيادة الصحة في حال كانت أقل من الحد الأقصى
		if(health < maxHealth){
			//لا تسمح للصحة بتجاوز الحد الأقصى
			health = Mathf.Min(maxHealth, health + amount);
			SendMessage("OnHealthIncrement",
					health,
					SendMessageOptions.DontRequireReceiver);
		}
	}

	//هل اللاعب ميت أم لا؟
	public bool IsDead(){
		return dead;
	}

	public int GetCurrentHealth(){
		return health;
	}
}
```

السرد 74: بريمج صحة اللاعب

عند إضافة البريمج لكائن اللاعب يمكننا تحديد القيمة الأولية للصحة عبر المتغير initialHealth والتي يقوم البريمج بالتأكد من أنها أكبر من الحد الأدنى وهو صفر وأقل من أو تساوي الحد الأقصى maxHealth. وهذا يمنع اللاعب من بدء اللعبة بمقدار صحة أكبر من الحد الأقصى كما يمنع أن يبدأ اللعبة ميتا. بطبيعة الحال فإن وصول قيمة صحة اللاعب إلى صفر أو أقل يعني موت اللاعب، آخذين بعين الاعتبار أن القيمة الفعلية لصحة اللاعب هي قيمة المتغير health والتي لا يمكن تعديلها مباشرة وإنما عبر الدّالتين ()InreaseHealth والتي تزيد من صحة اللاعب والدّالة ()DecreaseHealth التي تنقص من صحة اللاعب. استخدام هاتين الدّالتين ضروري من أجل التأكد من أنّ قيمة الصحة صالحة دائما أي ضمن الحدود وإرسال الرسائل المناسبة عند حدوث أي تغير حتى يتم إعلام البريمجات الأخرى

التي تهمها قيمة صحة اللاعب بالتغييرات أولا بأول. إضافة لذلك فإنّ استخدام الدّالة DecreaseHealth() ضروري لإعلام البريمجات الأخرى بموت اللاعب حال حدوثه وذلك عبر الرسالة OnPlayerDeath.

إذن فإن البريمج PlayerHealth يعطينا القدرة على إدارة صحة اللاعب ومعرفة موته حال حدث ذلك، كما أنّه المكان الوحيد الذي يمكن من خلاله تغيير صحة اللاعب زيادة ونقصانا. على الرغم من ذلك فإنّه لا يحدد بشكل واضح ما الذي يمكن أن يسبب نقصا أو زيادة في صحة اللاعب، ولهذا السبب سنحتاج لبريمج آخر يمكنه أن يستجيب للمؤثرات الخارجية التي تؤثر على صحة اللاعب وتقوم بإنقاص مقدارها بقيمة محددة. هذا البريمج هو PainfulDamageTaker وهو موضح في السرد 75.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(PlayerHealth))]
public class PainfulDamageTaker : MonoBehaviour {

	//متغير للوصول إلى بريمج صحة اللاعب
	PlayerHealth playerHealth;

	void Start () {
		playerHealth = GetComponent<PlayerHealth>();
	}

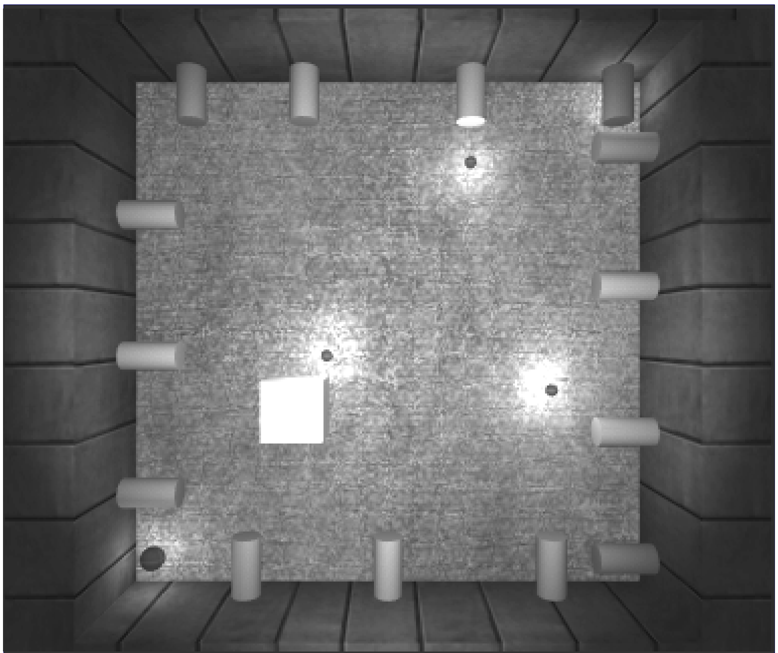
	void Update () {

	}

	//استقبال رسالة اصطدام الطلقة وإنقاص الصحة بناء
	//على قيمة الضرر المرفقة مع الرسالة
	void OnPainfulHit(int amount){
		playerHealth.DecreaseHealth(amount);
	}
}
```

السرد 75: البريمج الخاص باستقبال الطلقات وإنقاص صحة اللاعب بناء عليها

لاحظ أن كل ما احتجنا لفعله في هذا البريمج هو استقبال الرسالة OnPainfulHit إضافة إلى قيمة إنقاص الصحة المرفقة معها. تذكر أنّ هذه الرسالة ترسلها الطلقة وتحديدا البريمج PainfulProjectile عند اصطدامها بجسم ما. يستخدم هذا البريمج ذات القيمة المرفقة مع الرسالة OnPainfulHit عندما يستدعي الدّالة DecreaseHealth() من البريمج PlayerHealth، وهذا الأخير سبق وأعددناه ليقوم بكل الخطوات المطلوبة فيما يخص صحة اللاعب وتغير قيمتها. يمكنك الآن إضافة المكعب الذي يمثل اللاعب للمشهد وتحريكه محاولا تفادي الطلقات. سيبدو الشكل النهائي كما في الشكل 85.



الشكل 85: لقطة من اللعبة عند تشغيلها وتظهر المكعب الذي يتحكم به اللاعب إضافة للطلقات

ما ينقصنا الآن هو أن نعطي اللاعب دليلا مرئيا على حالته الصحية داخل اللعبة. فالإدارة الداخلية لمقدار الصحة كفيلة بمعرفة ما تبقى منها وما إذا كان اللاعب قد مات أم ليس بعد، بيد أنه من المهم أيضا أن نشارك هذه المعلومات مع اللاعب حتى يعرف حالته داخل اللعبة. أحد الخيارات هو إظهار قيمة الصحة كرقم على الشاشة، وهناك طرق أخرى لا حصر لها. أحد هذه الطرق وهو ما سنستعمله هو التمثيل اللوني لقيمة الصحة، حيث سنقوم بتغيير لون المربع من الأخضر والذي يعني الصحة الكاملة إلى الأحمر والذي يعني أن مقدار الصحة قليل. البريمج المسؤول عن هذه المهمة هو HealthColorChanger وهو موضح في السرد 76.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(PlayerHealth))]
public class HealthColorChanger : MonoBehaviour {

	//اللون الذي يمثل الصحة الكاملة
	public Color fullHealth = Color.green;

	//اللون الذي يمثل الموت
	public Color zeroHealth = Color.red;

	//متغير للوصول إلى بريمج صحة اللاعب
	PlayerHealth playerHealth;

	void Start () {
```

```
            playerHealth = GetComponent<PlayerHealth>();
            UpdateColor(playerHealth.GetCurrentHealth());
        }

        void Update () {

        }

        void OnHealthIncrement(int amount){
            UpdateColor(amount);
        }

        void OnHealthDecrement(int amount){
            UpdateColor(amount);
        }

        //قم بتحديث اللون على مكوّن التصيير اعتمادا على مقدار
        //القيمة الجديدة لصحة اللاعب
        void UpdateColor(int newHealth){
            //قم بتحويل مقدار الصحة من عدد صحيح إلى عدد كسري بحيث
            //تكون قيمته بين 1 وتعني الصحة الكاملة وصفر وتعني موت اللاعب
            float val = (float)newHealth /
                    (float) playerHealth.maxHealth;

            //قم بتطبيق اللون الجديد
            renderer.material.color =
                Color.Lerp(zeroHealth, fullHealth, val);
        }
}
```

السرد 76: بريمج يقوم باستيفاء لون كائن اللاعب بين قيمتين مختلفتين بناء على مقدار صحته

ما يقوم به هذا البريمج هو استقبال الرسالتين OnHealthIncrement و OnHealthDecrement التين يقوم PlayerHealth بإرسالهما عند تغير قيمة الصحة، ومن ثم يقوم بتحويل القيمة إلى نسبة مئوية بين صفر و 1. بعد ذلك يقوم باستدعاء الدالة ()Color.Lerp من أجل تغيير لون الكائن الحالي إلى القيمة الصحيحة بين لوني القيمة العظمى والقيمة الدنيا. إذا قمت بتجربة اللعبة بعد إضافة هذا البريمج للمكعب ستلاحظ أنه يبدأ أخضر اللون ومن ثم يميل للاحمرار مع تلقي الضربات ونقصان الصحة.

السؤال التالي الذي نحتاج للإجابة عليه هو: ماذا يحدث عندما يموت اللاعب؟ في الوقت الراهن لا يحدث أي شيء فعليا؛ وذلك لأننا لا نقوم بأي إجراء محدد عند وصول صحة اللاعب لصفر أو أقل. بطبيعة الحال فإن ما نرغب بفعله هو أن ننقص فرصة من فرص اللاعب ومن ثم نعيده للعبة من جديد بصحة كاملة للقيام بمحاولة أخرى. من أجل هذا علينا أن نعرف كيف نتعامل مع الرسالة OnPlayerDeath التي يقوم البريمج PlayerHealth بإرسالها عند موت اللاعب. ما يلزمنا عمله هو ربط هذه الرسالة بطريقة ما ببريمج على كائن خارجي مستقل يقوم بعدّ الفرص المتبقية للاعب. بمعنى آخر فإنّ هذا البريمج وهذا الكائن هو المسؤول عن إعادة بناء كائن اللاعب بعد موته وإنقاص الفرص المتبقية، وأخير فهو مسؤول

عن إنهاء اللعبة حين تنتهي كل فرص اللاعب المتوفرة. هذا البريمج هو LivesManager والموضح في السرد 77. الهدف من ربط هذا البريمج بكائن خارجي منطقي جدا وهو أنّه لا يجب أن يتم تدميره مع اللاعب بل يجب أن يكون دائما في المشهد حتى يتمكن من إحصاء عدد مرات موت اللاعب ومقارنة ذلك بما لديه من فرص. على سبيل المثال يمكننا أن نضيف هذا البريمج إلى كائن الكاميرا.

```
using UnityEngine;
using System.Collections;

public class LivesManager : MonoBehaviour {

	//عدد الفرص عند بداية اللعبة
	public int startLives = 3;

	//المتغير الداخلي لإحصاء عدد الفرص
	int lives;

	void Start () {
		//يلزم على الأقل فرصة واحدة عند البداية
		if(startLives > 0){
			lives = startLives;
		} else {
			lives = 1;
		}
	}

	void Update () {
	}

	public void GiveLife(){
		lives++;
		SendMessage("OnLifeGained",
				lives, //العدد الجديد للفرص
				SendMessageOptions.DontRequireReceiver);
	}

	public void TakeLife(){
		lives--;
		if(lives == 0){
			//تم فقدان الفرصة الأخيرة
			//يجب إعلام البريمجات الأخرى ليقوم أحدها بالتعامل مع هذا الأمر
			SendMessage("OnAllLivesLost",
				SendMessageOptions.DontRequireReceiver);
		} else {
			//تم فقدان فرصة لكن ما زال هناك غيرها
			//قم بإرسال رسالة ليتم التعامل مع هذا الأمر
			SendMessage("OnLifeLost",
				lives, //عدد الفرص المتبقية
				SendMessageOptions.DontRequireReceiver);
		}
	}
}
```

السرد 77: البريمج الخاص بالتحكم بعدد الفرص المتبقية للاعب

يقوم هذا البريمج بالتعامل مع عدد الفرص بطريقة مشابهة لما يقوم به البريمج PlayerHealth مع صحة اللاعب، حيث لدينا متغير يحدد العدد الأولي للفرص ومن ثم فإنّ التحكم بعددها زيادة أو نقصانا يتم فقط عبر الدّالتين ()GiveLife و ()TakeLife وهذه الأخيرة تتميز بأنها يمكن أن ترسل رسالتين وهما OnLifeLost والتي تدل على فقدان فرصة مع وجود غيرها لدى اللاعب و OnAllLivesLost والتي تدل على فقدان جميع الفرص. ما علينا فعله الآن هو أن نقوم باستدعاء الدّالة ()TakeLife في كل مرّة تصل فيها صحة اللاعب لقيمة أقل من أو مساوية للصفر. أي أننا نحتاج لاستقبال الرسالة OnPlayerDeath وإرسال الرسالة TakeLife إلى البريمج LivesManager. من سيتولى هذه المهمة هو البريمج PlayerDeathReporter والموضح في السرد 78.

```
using UnityEngine;
using System.Collections;

public class PlayerDeathReporter : MonoBehaviour {

	//LivesManager متغير للوصول إلى الكائن المحتوي على البريمج
	LivesManager manager;

	void Start () {
		manager = FindObjectOfType<LivesManager>();
	}

	void Update () {

	}

	//قم بإنقاص فرص اللاعب في كل مرة يموت فيها
	void OnPlayerDeath(int deathHealth){
		if(manager != null){
			manager.TakeLife();
		}
	}
}
```

السرد 78: البريمج الخاص بإرسال الرسالة TakeLife في كل مرة يموت فيها اللاعب

البريمج بسيط جدا كما ترى ولا يحتاج كثيرا من الشرح. علينا أن نتذكر هنا أنّ البريمج LivesManager يقوم بالتحكم بعدد الفرص المتبقية للاعب، لكنه لا يتحكم بحالة اللعبة، وبالتالي لا يملك رد فعل معينا في حال خسر اللاعب كل الفرص المتاحة. لأجل ذلك سنحتاج لبناء آلية لتدمير وإعادة إنشاء كائن اللاعب أي المكعب والذي سبق وأن قمنا بعمله على شكل قالب استعدادا لهذه اللحظة التي سنحتاج فيها لهذا القالب. البريمج PlayerSpawn يتولى مهمة تدمير كائن اللاعب حال موته وإعادة إنشاء كائن جديد في حال كان هناك المزيد من الفرص المتوفرة. هذا البريمج موضح في السرد 79.

```
using UnityEngine;
using System.Collections;

public class PlayerSpawn : MonoBehaviour {
```

```

    //القالب الخاص بكائن اللاعب
    public GameObject playerPrefab;

    //الثواني التي يجب انتظارها ما بين تدمير الكائن وإعادة إنتاجه في الفرصة التالية
    public int spawnDelay = 3;

    //متغير لتخزين كائن اللاعب الحالي
    GameObject currentInstance;

    void Start () {
        SpawnPlayer();
    }

    void Update () {

    }

    //تمت خسارة فرصة
    void OnLifeLost(int remainingLives){
        //قم بتدمير كائن اللاعب واستدعاء إعادة الإنشاء بعد التأخير المطلوب
        Destroy(currentInstance);
        Invoke("SpawnPlayer", spawnDelay);
    }

    //انتهت اللعبة: قم بتدمير كائن اللاعب ولا تنشئ غيره
    void OnAllLivesLost(){
        Destroy(currentInstance);
    }

    void SpawnPlayer(){
        currentInstance =
            (GameObject) Instantiate(playerPrefab);
    }
}
```

السرد 79: البريمج الخاص بتدمير وإعادة إنشاء كائن اللاعب بناء على موته واستخدامه لفرصة أخرى

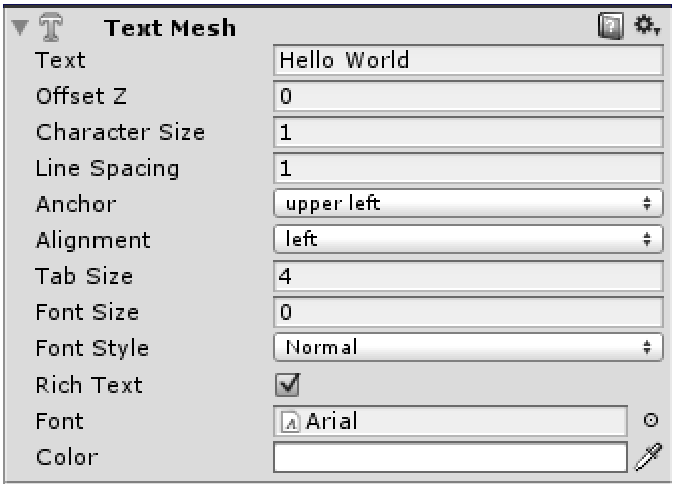
ما يحتاجه هذا البريمج ليعمل هو قالب لإنشاء كائن اللاعب من خلاله بالإضافة إلى عدد من الثواني التي يجب انتظارها قبل إعادة إنشاء كائن اللاعب مرة أخرى. إنشاء اللاعب يتم من خلال استدعاء الدّالة ()SpawnPlayer والتي تقوم بعمل نسخة من قالب اللاعب وتحتفظ بمرجع لها عبر قيمة المتغير currentInstance. أهمية هذا المرجع تكمن في الحاجة لتدمير كائن اللاعب في وقت لاحق حين يموت، وبالتالي سنحتاج لاستقبال رسائل موت اللاعب وفقدانه للفرص من البريمج LivesManager والذي يفترض وجوده على نفس الكائن. الرسالتان المهمتان بالنسبة لهذا البريمج هما OnLifeLost والتي يقوم بناء عليها بتدمير كائن اللاعب الحالي وإنشاء واحد آخر جديد عبر استدعاء ()SpawnPlayer وذلك مع تأخير يعادل spawnDelay من الثواني. الرسالة الأخرى المهمة هي OnAllLivesLost والتي يقوم بناء عليها بتدمير كائن اللاعب دون إنشاء واحد جديد، أي دون استدعاء ()SpawnPlayer.

كما سبق واحتجنا لتمثيل مرئي لمقدار صحة اللاعب وذلك عن طريق لون المكعب، سنحتاج أيضا

لتمثيل مرئي لعدد الفرص المتبقية. الطريقة الأبسط هي إظهار عددها كنص على الشاشة، وهذا يقودنا لاستخدام كائن جديد هو 3D Text والذي يمكننا وضعه في أي مكان في المشهد ويظهر لنا نصا مكتوبا بحروف ثلاثية الأبعاد. ما سيلزمنا في حالتنا هذه وهو وضع هذا الكائن أمام الكاميرا مباشرة.

لإضافة كائن نص ثلاثي الأبعاد 3D Text إلى المشهد اذهب إلى Game Object > Create Other > 3D Text. يمكنك بعدها تحريك النص وتحجيمه كأي كائن آخر في المشهد حيث ينبغي وضعه أمام الكاميرا مباشرة ليكون مرئيا للاعب

الشكل 86 يظهر أهم متغيرات هذا الكائن كما تظهر في شاشة الخصائص.



الشكل 86: المتغيرات الخاصة بكائن النص ثلاثي الأبعاد كما تظهر في شاشة الخصائص

معظم الخصائص تبدو واضحة ومعتادة لمن يتعامل مع نصوص الحاسب في برامج أخرى. سيكون تركيزنا منصبا على الخانة Text والتي سنحتاج لتغيير قيمتها عن طريق بريمج ما بحيث يعرض هذا الكائن عدد الفرص المتبقية للاعب. لنقم أولا بوضع الكائن في مكان مناسب وليكن أعلى الشاشة مثلا، ومن الجيد أن نختار له قيمة أولية كالرقم صفر مثلا. علينا بعدها أن نكتب بريمجا يقرأ عدد الفرص المتبقية للاعب ويعرضها كنص على هذا الكائن. هذا البريمج هو LivesCounterHandler الموضح في السرد 80.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(LivesManager))]
public class LivesCounterHandler : MonoBehaviour {

	//مرجع لكائن النص ثلاثي الأبعاد
	public TextMesh display;

```

```
10.      //مرجع لبريمج إدارة الفرص
11.      LivesManager lManager;
12.
13.      void Start () {
14.            lManager = GetComponent<LivesManager>();
15.            //مبدئيا سنقوم بإظهار عدد الفرص الأولي الذي تم تحديده
16.            display.text = lManager.startLives.ToString();
17.      }
18.
19.      void Update () {
20.
21.      }
22.
23.      void OnLifeGained(int newLives){
24.            //ازداد عدد الفرص، بالتالي علينا تحديث النص
25.            display.text = newLives.ToString();
26.      }
27.
28.      void OnLifeLost(int remainingLives){
29.            //قل عدد الفرص، بالتالي علينا تحديث النص
30.            display.text = remainingLives.ToString();
31.      }
32.
33.      void OnAllLivesLost(){
34.            //خسر اللاعب كل الفرص
35.            //لذلك سنكتب عبارة "انتهت اللعبة"
36.            display.text = "Game Over";
37.      }
38. }
```

السرد 80: البريمج الخاص بالتحكم بعرض الفرص المتبقية للاعب على شكل نص

لاحظ أننا نصل لكائن النص ثلاثي الأبعاد عن طريق متغير من نوع TextMesh، لاحظ أيضا أن هذا البريمج يحتاج لوجود البريمج LivesManager وذلك لأنه سيكون المصدر الرئيسي للمعلومات التي سيقوم بعرضها على شكل نص ثلاثي الأبعاد. بعد إضافة هذا البريمج لكائن الكاميرا (وهو الكائن الذي أفترض أنك أضفت البريمج LivesManager إليه) علينا أن نقوم بسحب كائن النص ثلاثي الأبعاد من الهرمية إلى داخل خانة المتغير display في شاشة خصائص هذا البريمج. لاحظ أنه يمكننا تغيير النص المعروض عن طريق المتغير display.text وبالتالي نقوم في البداية داخل الدّالة ()Start بعرض قيمة startLives. بما أن المتغير display.text هو عبارة عن متغير نصّي string بينما startLives هو عدد صحيح int، ينبغي علينا استخراج قيمة هذا الرقم على شكل نص حتى نتمكن من إسنادها للمتغير النصي. هذه العملة تتم عبر استدعاء الدّالة ()ToString كما في السطر 16. بعد عرض القيمة الأولية كل ما يتبقى لنا هو أن نتابع أي تغيير على عدد الفرص عن طريق استقبال الرسالتين OnLifeGaind و OnLifeLost وذلك لتغيير النص المعروض. أخيرا علينا أن ننتبه للرسالة OnAllLivesLost وذلك حتى نعرض نصا يخبر اللاعب بنهاية اللعبة وهو Game Over.

الموضوع الأخير الذي سنناقشه في هذا الفصل هو نقاط اللاعب. فنحن الآن نمتلك لعبة متكاملة

القواعد وكل ما ينقصنا هو تقييم أداء اللاعب عن طريق النقاط التي يحصل عليها. بما أن طبيعة اللعبة تتطلب من اللاعب النجاة لأطول وقت ممكن من الطلقات، فإنّه من المنطقي أن نستخدم الوقت الذي يمكن للاعب أن يبقى خلاله على قيد الحياة كمعيار لمهارته في اللعب. سنحتاج كما في حالة الفرص إلى نص ثلاثي الأبعاد لعرض النقاط وسنضيفه أسفل الشاشة، كما سنحتاج لبريمج يعد الثواني منذ بداية اللعبة حتى نهايتها ويزيد النقاط كل ثانية. كما هو الحال مع البريمجات الأخرى المتعلقة بحالة اللعبة علينا أن نضيف بريمج النقاط إلى كائن دائم وليكن الكاميرا أيضا. السرد 81 يوضح البريمج ScoreCounter والذي يتولى مهمة إحصاء نقاط اللاعب وعرضها.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(LivesManager))]
public class ScoreCounter : MonoBehaviour {

	//متغير اختياري لعرض عدد النقاط على شكل نص
	public TextMesh display;

	LivesManager lManager;

	//العداد الداخلي للنقاط
	int score = 0;

	void Start () {
		lManager = GetComponent<LivesManager>();
		//قم بزيادة النقاط بمقدار نقطة كل ثانية
		InvokeRepeating("IncrementScore", 1, 1);
	}

	void Update () {

	}

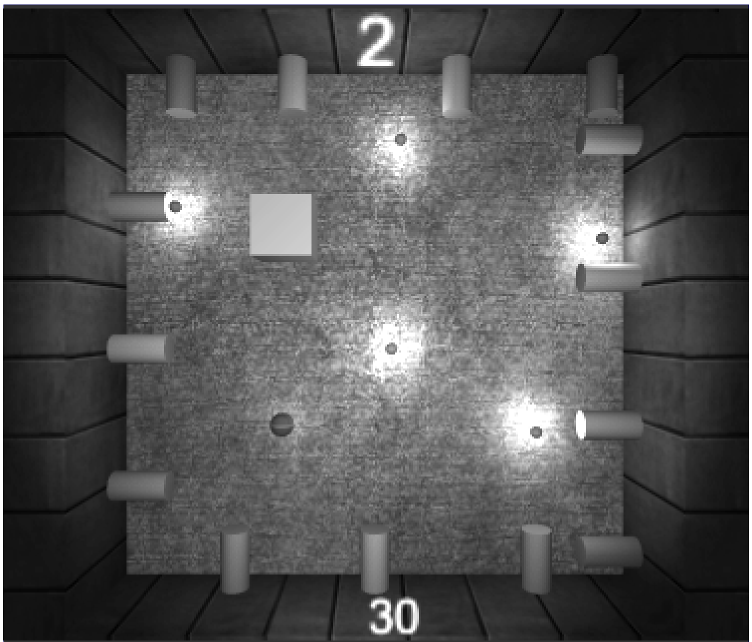
	void IncrementScore(){
		score++;
		if(display != null){
			display.text = score.ToString();
		}
	}

	void OnAllLivesLost(){
		//انتهت اللعبة، إذن علينا التوقف عن إحصاء النقاط
		CancelInvoke("IncrementScore");
	}
}
```

السرد 81: بريمج يعمل على زيادة نقاط اللاعب كل ثانية وعرضها على كائن نص ثلاثي الأبعاد

يعتمد هذا البريمج على استدعاء الدّالة ()IncrementScore مرة واحدة كل ثانية وذلك عن طريق استدعاء ()InvokeRepeating عند بداية اللعبة. بالتالي فإنه عند كل ثانية تمر من وقت اللعب تتم زيادة

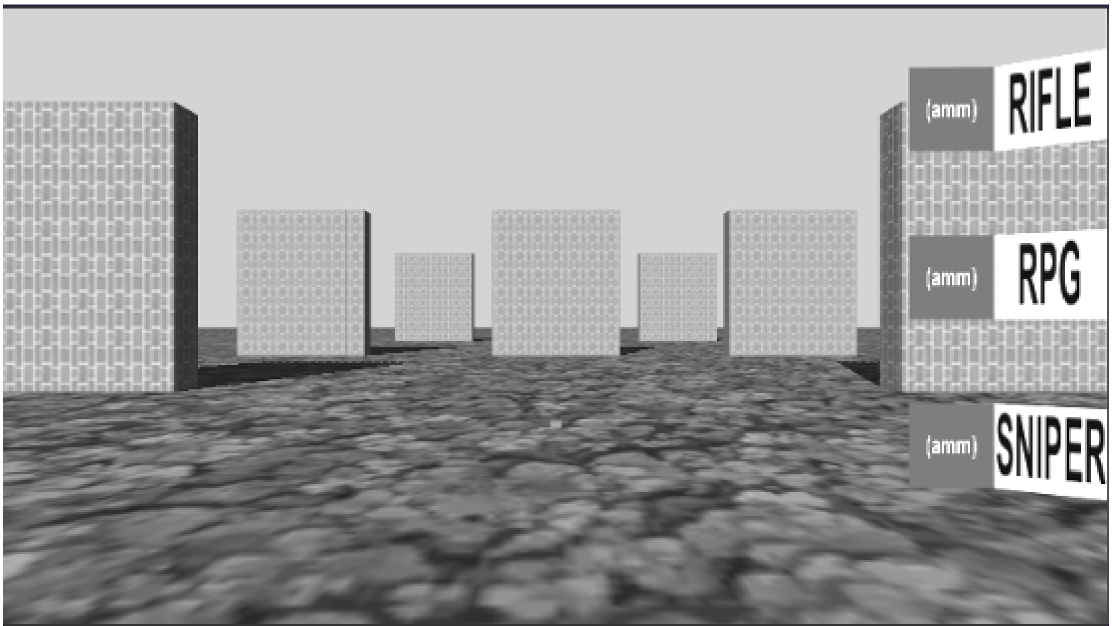
عدد النقاط إضافة إلى تحديث النص الذي يعرضها على الشاشة. عند استقبال الرسالة OnAllLivesLost يجب التوقف عن إحصاء النقاط وذلك لأن اللعبة انتهت. إيقاف الاستدعاء المتكرر للدّالة IncrementScore() يتم عبر استدعاء CancelInvoke() وإعطائها اسم الدالة التي نرغب بإيقاف استدعائها كما في السطر 34. يمكنك مشاهدة النتيجة النهائية للعبة التي أنشأناها في هذا الفصل في الشكل 87، كما يمكنك تجربتها في المشهد scene21 في المشروع المرفق.



الشكل 87: المشهد النهائي للعبة حيث يظهر عدد الفرص في النص أعلى الشاشة والنقاط في النص أسفل الشاشة

الفصل الرابع: الأسلحة والذخيرة وإعادة التعبئة

في كثير من الألعاب التي تحتوي على ميكانيكية التصويب، يكون من الممكن عادة أن يحمل اللاعب عددا من الأسلحة في آن واحد ويقوم بالتبديل بينها حسب الحاجة. إضافة لذلك تحتوي هذه الأسلحة غالبا على كمية محدودة من الذخيرة التي تحتاج لإعادة التعبئة من وقت لآخر. هذه الذخيرة تكون مقسمة في أحيان كثيرة على مخازن يكون كل منها ذو سعة محددة، مما يضيف ميكانيكية أخرى هي إعادة التعبئة أو التذخير، والتي تنتشر بشكل كبير في ألعاب التصويب من منظور الشخص الأول.
سنقوم في هذا الفصل بتطبيق جميع وظائف الأسلحة المذكورة، ولنبدأ مع مشهد شبيه بما في الشكل 88. في هذا المثال لن يكون اللاعب قادرا على الحركة، بل سيقوم بالتأشير بالفأرة على الهدف وإطلاق النار عليه باستخدام زرها الأيسر. إضافة لذلك سنقوم باستخدام لوحة المفاتيح وتحديدا مفاتيح الأرقام للتبديل بين الأسلحة. أخيرا لاحظ وجود نصوص ثلاثية الأبعاد على يمين الشاشة والتي سنستخدمها لعرض كمية الذخيرة المتبقية في كل سلاح إضافة لتقدم عملية إعادة التعبئة.



الشكل 88: مشهد خاص بتجربة تبديل الأسلحة وإعادة تعبئتها

جدير بالذكر أن الجدران التي تراها في الشكل 88 تم إنشاؤها باستخدام وحدات بنائية قابلة للهدم شبيهة بتلك التي استخدمناها في الفصل السادس من الوحدة الرابعة. رغم ذلك فمن الأفضل استخدام قالب جديد خاص بهذه الوحدات البنائية يختلف عن ذلك الذي استخدمنا سابقا؛ والسبب أننا سنقوم بإجراء بعض التعديلات عليه لناسب احتياجاتنا الحالية. من ميزات الوحدات البنائية هي إمكانية تعديل عدد كبير من الكائنات من مكان واحد، لذا ما سنقوم به هو نسخ القالب الحالي المسمى ReturnableBrick وتسمية النسخة الجديدة ShootableBreak ومن ثم استخدامها لبناء المشهد الجديد. سنعود لاحقا لهذا القالب لإجراء التعديلات اللازمة، لكن علينا الآن أن نضيف كائن اللاعب ولنسمّه player. سنضيف هذا الكائن في المشهد في نفس موقع الكاميرا التي ننظر من خلالها وسنقوم باستخدامه كموقع لإطلاق النار. هذا يعني بطبيعة الحال أنّ كائن اللاعب يجب أن ينظر للأمام باتجاه الجدران، أي أن محور z الخاص به يجب أن يشير باتجاهها. أخيرا سنضيف ثلاثة أبناء هي عبارة عن كائنات فارغة سنستخدمها ككائنات للأسلحة بالتالي سنسميها بأسماء تمثلها: Rifle, RPG, Sniper.

تمتلك الأسلحة المذكورة خصائص متشابهة، مثل إمكانية أطلاق النار من خلالها بطبيعة الحال، إضافة إلى حاجتها للذخيرة وهكذا. على الجانب الآخر فإن كل واحد من هذه الأسلحة لديه طريقته الخاصة في إطلاق النار: فالقناص (sniper) يطلق في كل مرة طلقة واحدة عالية الدقة في الإصابة، بينما الرشاش (rifle) فيقوم بإطلاق عدد أكبر من الطلقات في وقت قصير، وأخيرا فإنّ قاذفة RPG تطلق قذيفة واحدة في كل مرة. هذا يعني أن علينا فصل الوظائف الأساسية المشتركة عن الوظائف الخاصة بكل سلاح وكيفية إطلاق النار فعليا من خلاله. بداية سنحتاج للبريمج GeneralWeapon والذي يحتوي على الوظائف الأساسية المشتركة بين الأسلحة، وهو موضح في السرد 82.

```
using UnityEngine;
using System.Collections;

public class GeneralWeapon : MonoBehaviour {

	//عدد مخازن الذخيرة المتبقية
	public int magazineCount = 3;

	//السعة التخزينية من الطلقات لكل مخزن ذخيرة
	public int magazineCapacity = 30;

	//عدد الطلقات المتبقية في المخزن الحالي
	public int magazineSize = 30;

	//كم من الوقت نحتاج لإعادة التعبئة؟
	public float reloadTime = 3;

	//كم عدد الطلقات التي يستطيع السلاح إطلاقها في الثانية الواحدة؟
	public float fireRate = 3;

	//true إذا كانت قيمة هذا المتغير هي
	//فذلك يعني أن لا حاجة لرفع الإصبع عن الزناد بين عمليتي الإطلاق المتتاليتين
	public bool automatic = false;

	//كمية الطلقات التي تنقص من مخزن الذخيرة الحالي مع كل عملية إطلاق للنار
	//يجب ألا يتجاوز هذا الرقم السعة التخزينية لمخزن الذخيرة
	public int ammoPerFiring = 1;

	//هل هذا السلاح هو المستخدم حاليا من قبل اللاعب؟
	public bool inHand = false;

	//مؤقت داخلي لحساب وقت الانتظار بين عمليات الإطلاق
	float lastFiringTime = 0;

	//متغير داخلي لتقدم عملية إعادة التعبئة
	float reloadProgress = 0;

	//متغير داخلي لمعرفة ما إذا كان اللاعب حاليا يضغط على الزناد
	bool triggerPulled = false;

	void Start () {

	}

	void Update () {
		//إذا كان اللاعب يستخدم هذا السلاح حاليا وعملية إعادة التعبئة لم تتم بعد
		//علينا في هذه الحالة أن نقوم بزيادة مقدار تقدم العملية
		if(inHand && reloadProgress > 0){
			reloadProgress += Time.deltaTime;
			if(reloadProgress >= reloadTime){
				//اكتملت إعادة التعبئة
				//قم بالتخلص من المخزن الحالي
				//وتركيب واحد جديد
				magazineSize = magazineCapacity;
```

```
                    magazineCount--;
                    reloadProgress = 0;
                    SendMessage("OnReloadComplete",
                          SendMessageOptions.DontRequireReceiver);
              }
        }
  }

  public void Fire(){
        //هناك عدو شروط قبل إطلاق النار وهي أن يكون السلاح حاليا بيد اللاعب
        //وألا يكون قيد إعادة التعبئة، كما ينبغي التأكد من مرور الفترة الزمنية المحددة
        //بين عمليتي الإطلاق المتتاليتين وأن يكون السلاح إمّا أوتوماتيكيا أو أنّ اللاعب
        //قد قام برفع إصبعه عن الزناد بعد آخر عملية إطلاق نار تمت
        if(inHand && reloadProgress == 0 &&
              (automatic || !triggerPulled) &&
              Time.time - lastFiringTime > 1 / fireRate){
              //هل يحتوي المخزن الحالي على عدد كاف من الطلقات؟
              if(magazineSize >= ammoPerFiring){
                    //نعم، بالتالي تتم عملية الإطلاق عبر إنقاص الذخيرة
                    //وتحديد وقت الإطلاق الأخير، إضافة إلى
                    //OnWeaponFire إرسال الرسالة
                    magazineSize -= ammoPerFiring;
                    lastFiringTime = Time.time;
                    triggerPulled = true;
                    SendMessage("OnWeaponFire",
                          SendMessageOptions.DontRequireReceiver);

                    //إن كانت الذخيرة المتبقية في المخزن غير كافية يجب إعادة التعبئة
                    if(magazineSize < ammoPerFiring){
                          Reload();
                    }

              } else {
                    //لا توجد ذخيرة كافية للإطلاق، يجب إعادة التعبئة
                    Reload();
              }
        }
  }

  public void ReleaseTrigger(){
        triggerPulled = false;
  }

  public void Reload(){
        //قم بالتأكد من عدم وجود عملية تعبئة قيد التنفيذ
        if(reloadProgress == 0){
              //قم بالتأكد من وجود عدد كاف من المخازن وأنّ
              //المخزن الحالي ليس ممتلئا
              if(magazineCount > 0 &&
                    magazineSize < magazineCapacity){
                    //قم ببدء عملية إعادة التعبئة
                    reloadProgress = Time.deltaTime;
                    SendMessage("OnReloadStart",
                          SendMessageOptions.DontRequireReceiver);
              }
```

```
			}
		}

		//تعيد هذه الدّالة مقدار التقدم الحاصل حاليا في إعادة التعبئة
		public float GetReloadProgress(){
			return reloadProgress / reloadTime;
		}
}
```

السرد 82: البريمج الخاص بالوظائف الأساسية المشتركة بين جميع الأسلحة

تقوم المتغيرات الثلاث الأولى بإدارة كمية الذخيرة المتوفرة في السلاح. الفرق بين magazineCapacity و magazineSize هو أن الأول رقم ثابت يعبر عن الحد الأقصى من الطلقات الذي يمكن أن يسعه مخزن واحد، بينما يمثل الآخر عدد الطلقات المتبقية في المخزن المثبت بالسلاح حاليا. هذا العدد ينقص مع كل عملية إطلاق نار بمقدار يساوي ammoPerFiring أي عدد الطلقات التي تخرج من السلاح حين استخدامه لمرة واحدة. بالإضافة لمكية الذخيرة هناك متغيرات للتحكم بسرعة الإطلاق مثل reloadTime والذي يحدد الوقت اللازم لإتمام عملية إعادة تعبئة السلاح (أي استبدال المخزن الحالي بآخر جديد). إضافة لذلك لدينا المتغير fireRate والذي يحدد كم مرة يمكن إطلاق النار من السلاح في الثانية الواحدة. المتغيران الداخليان lastFiringTime و reloadProgress يعملان جنبا إلى جنب مع fireRate و reloadTime من أجل حساب التوقيتات بشكل صحيح. الأمر الآخر المهم هو تحديد ما إذا كان السلاح أوتوماتيكيا، والذي يعني قدرة السلاح على الاستمرار في إطلاق النار طالما لم يرفع اللاعب إصبعه عن الزناد. هذه العملية تتم إدارتها عبر المتغيرين automatic و triggerPulled. أخيرا لدينا المتغير inHand والذي يحدد ما إذا كان هذا السلاح هو الموجود حاليا بين يدي اللاعب، وهذا المتغير هو بمثابة المفتاح الرئيسي لجميع الوظائف الأخرى؛ حيث لا يمكن أن يتم إطلاق للنار أو إعادة للتعبئة طالما لم يكن السلاح بين يدي اللاعب ابتداءا.

الدّالتان ()Fire و ()ReleaseTrigger مرتبطتان ببعضهما؛ حيث إنّه في حالة السلاح غير الأوتوماتيكي فإنّه لا بد من استدعاء ()ReleaseTrigger مرة على الأقل بعد كل مرة يتم فيها استدعاء ()Fire وذلك حتى تصبح عملية الإطلاق التالية ممكنة الحدوث. عند إطلاق النار عبر استدعاء الدّالة ()Fire، فإنّ هذه الأخيرة عليها أن تتأكد من عدة أمور وهي:

- أنّ اللاعب يحمل السلاح الحالي بين يديه (قيمة inHand هي true)
- أنّ السلاح ليس قيد إعادة التعبئة حاليا (قيمة reloadProgress هي صفر)
- أنّ السلاح أوتوماتيكي أو أنّه تم رفع الإصبع عن الزناد بعد آخر عملية إطلاق (قيمة automatic هي true أو قيمة triggerPulled هي false)
- وجود ذخيرة كافية في السلاح (قيمة magazineSize أكبر من أو تساوي قيمة

(ammoPerFiring

في حال تحقق جميع الشروط أعلاه، فإنّ البريمج يقوم بإرسال الرسالة OnWeaponFire بالإضافة إلى تغيير قيمة triggerPulled إلى true، ومن ثم يقوم أخيرا بإنقاص القيمة ammoPerFiring من مجموع الطلقات في المخزن الحالي magazineSize. إذا أصبح العدد الجديد للطلقات في المخزن الحالي أقل من عدد الطلقات اللازمة لإطلاق النار فإنّ البريمج يقوم تلقائيا باستبدال المخزن عن طريق استدعاء الدّالة ()Reload. مهمة هذه الدّالة هو أن تقوم ببدء عملية إعادة التعبئة وليس أن تقوم بها مباشرة ومرة واحدة، حيث يمث المتغير reloadProgress الوقت المنقضي منذ بدأت عملية إعادة التعبئة. تقوم الدّالة ()Reload أولا بفحص قيمة المتغير reloadProgress حيث يفترض أن تكون قيمتها صفرا إذا لم تكن عملية إعادة التعبئة بدأت بعد. بعد ذلك ينبغي التأكد من وجود مخزن إضافي واحد على الأقل غير ذلك الموجود على السلاح حاليا؛ لذا تقوم ()Reload بفحص قيمة magazineCount. إضافة إلى ذلك تقوم الدّالة أيضا بمقارنة قيمتي magazineSize و magazineCapacity، حيث أنّ تساوي هاتين القيمتين يعني أنّ المخزن الحالي ممتلئ بالتالي لا حاجة لاستبداله من الأساس. أخيرا بعد التحقق من كافة الشروط يتم تغيير قيمة المتغير reloadProgress إلى Time.deltaTime مما يؤدي إلى تجميع قيم Time.deltaTime أثناء تصيير الإطارات اللاحقة عن طريق الدّالة ()Update. تجميع هذه القيم يتم داخل المتغير reloadProgress، حتى إذا وصلت قيمته لقيمة أكبر من أو تساوي الوقت اللازم لإعادة التعبئة reloadTime عنى هذا أنّ عملية إعادة التعبئة قد اكتملت بالتالي يتم إنقاص عدد المخازن المتبقية بمقدار واحد وتغيير كمية الذخيرة في المخزن الحالي magazineSize إلى السعة الكاملة .magazineCapacity

آخر دالة سنتناولها من هذا البريمج هي ()GetReloadProgress. إذا كان السلاح الحالي قيد إعادة التعبئة فإنّ هذه الدّالة يفترض أن تعيد مقدار التقدم في التعبئة على شكل قيمة عدد كسري بين صفر وواحد. إرجاع القيمة صفر يعني أن السلاح ليس قيد إعادة التعبئة حاليا. هذه القيمة قد تكون مهمة لاستخدامها في مؤشرات تقدم أو لتحريك نموذج معين حسب تقدم إعادة التعبئة. الشكل 89 يظهر البريمج GeneralWeapon كما يظهر في شاشة الخصائص الخاصة بكل سلاح. من المهم هنا ذكر حقيقة أن القيمة fireRate لا تؤثر كثيرا في الأسلحة غير الأوتوماتيكية. لذا فبالرغم من حقيقة أنّ كلا من سلاح القناص Sniper وقاذفة RPG تم ضبطهما على قيم عالية إلّا أنّ هذا لن يكون ملاحظا لأن على اللاعب أن يفلت الزناد (زر الفأرة الأيسر) ويعيد الضغط عليه وهي عملية تأخذ وقتا، إضافة لأن قاذفة RPG لا تمتلك مخزنا بل يجب تركيب كل قذيفة لوحدها قبل إطلاقها.



الشكل 89: إعداد خصائص البريمج GeneralWeapon للأسلحة الثلاث المختلفة

لو اعتبرنا أنّ البريمج GeneralWeapon هو وحدة معالجة عملية إطلاق النار، فإنّ هذه الوحدة ستحتاج لمدخلات ومخرجات حتى يكتمل نظام الأسلحة الذي نسعى لبنائه. المدخلات ستكون عبارة عن التصويب نحو الهدف ومن ثم استدعاء الدّالة ()Fire بناء على مدخلات اللاعب عبر الفأرة. على الجانب الآخر فإنّ المخرجات ستكون عبارة عن بريمجات تستقبل الرسالة OnWeaponFire وتقوم بترجمتها لعملية إطلاق نار فعلية ذات تأثير على المشهد. لنبدأ أولا مع نظام المدخلات كونه مشتركا بين الأسلحة الثلاث بخلاف نظام المخرجات. للتذكير فإننا سنستخدم الفأرة للتصويب وإطلاق النار وبالتالي سنحتاج لبريمج ينظر مباشرة لموقع مؤشر الفأرة في المشهد. هذا البريمج هو MousePointerFollower الموضح في السرد 83 والذي سنقوم بإضافته لكائن اللاعب.

```
using UnityEngine;
using System.Collections;

public class MousePointerFollower : MonoBehaviour {

        //كائن يستخدم كعلامة لموقع الهدف داخل المشهد
        public Transform targetMarker;

        void Start () {
                //قم بإخفاء العلامة خلف اللاعب
                targetMarker.position =
                        transform.position - Vector3.forward;
        }

        void Update () {
                //حاول العثور على ما يشير إليه مؤشر الفأرة داخل المشهد
                Ray camToMouse =
                        Camera.main.ScreenPointToRay (Input.mousePosition);

                RaycastHit hit;
                if(Physics.Raycast(camToMouse, out hit, 500)){
                        //تم العثور على الكائن، قم بالنظر لموقع المؤشر فوقه
                        transform.LookAt(hit.point);
                        //قم بتحريك العلامة إلى موقع المؤشر
```

```
                    targetMarker.position = hit.point;
                    //قم بتحريك العلامة قليلا باتجاه اللاعب حتى تصبح مرئية
                    targetMarker.LookAt(transform.position);
                    targetMarker.Translate(0, 0, 0.1f);
            } else {
                    //لا يوجد هدف تحت مؤشر الفأرة، قم باختيار نقطة بعيدة
                    //والنظر إليها
                    transform.LookAt(camToMouse.GetPoint(500));
                    //قم بإخفاء العلامة
                    targetMarker.position =
                        transform.position - Vector3.forward;
            }
    }

}
```

السرد 83: بريمج بجعل الكائن ينظر بشكل دائم لموقع مؤشر الفأرة داخل المشهد

سنستخدم في هذا المثال ضوءا نقطيا أحمر اللون كعلامة، ولجعله يبدو كمؤشر ليزر سنقوم بتقليل نصف قطر محيط الإضاءة وزيادة كثافتها. بعد ذلك كل ما علينا هو سحب هذا الضوء للمتغير targetMarker. يقوم البريمج مبدئيا بإخفاء العلامة خلف كائن اللاعب والذي بدوره يقع في موقع الكاميرا مما يجعله يختفي. نقوم بعدها خلال كل دورة تحديث بإرسال شعاع من موقع اللاعب مرورا بمؤشر الفأرة إلى داخل المشهد. نقوم بعدها بفحص أي تصادم بين الشعاع وأي كائن في المشهد، فإذا وُجد هذا الكائن فإننا نقوم بوضع العلامة على نقطة اصطدام الشعاع به، مما يجعل هذه العلامة دائما تحت مؤشر الفأرة. حتى تكون العلامة مرئية نقوم بتحريكها قليلا نحو اللاعب حتى لا يغطيها الكائن الهدف. إضافة إلى ذلك علينا أن نضمن أنّ كائن اللاعب نفسه ينظر دائما في اتجاه مؤشر الفأرة، والذي يؤدي لأن تكون جميع الأسلحة أيضا مصوبة في ذلك الاتجاه كونها مضافة كأبناء لكائن اللاعب. من ناحية أخرى إن لم يوجد أي كائن تحت مؤشر الفأرة فإننا نوجه نظر اللاعب لنقطة بعيدة بمقدار 500 متر عن موقع اللاعب ولكن على امتداد الشعاع بين اللاعب ومؤشر الفأرة، بيد أننا في هذه الحالة سنخفي العلامة خلف اللاعب مرة أخرى.

بعد الانتهاء من تصويب نظر اللاعب وأسلحته إلى الوجهة الصحيحة، علينا أن نتعامل مع أوامر الإدخال من تبديل الأسلحة وإطلاق النار وإعادة التعبئة. سيقوم البريمج WeaponController الموضح في السرد 84 بالتعامل مع هذه الأوامر. هذا البريمج يجب أن تتم إضافته أيضا لكائن اللاعب.

```
using UnityEngine;
using System.Collections;

public class WeaponController : MonoBehaviour {

        //مصفوفة تحتوي على الأسلحة المتوفرة
        public GeneralWeapon[] weapons;

        //الموقع الخاص بالسلاح الذي يحمله اللاعب ابتداء
```

```
    public int initialWeapon = -1;

    //الموقع الخاص بالسلاح الذي يحمله اللاعب حاليا
    int currentWeapon;

    void Start () {
        //قم بضبط قيمة السلاح الحالي إلى السلاح الأولي
        //الذي تم اختياره عبر نافذة الخصائص
        currentWeapon = initialWeapon;
        //قم بتحديث قيم المتغير inHand لجميع الأسلحة
        RefreshInHandValues();
    }

    void Update () {
        UpdateSwitching();
        UpdateShooting();
    }

    void UpdateSwitching(){
        //قم بتحويل قيمة المفتاح "1" من لوحة المفاتيح إلى قيمة رقمية
        //نتحدث هنا عن المفتاح الذي يقع فوق الأحرف وليس على لوحة الأرقام اليمنى
        int keyCode = (int)KeyCode.Alpha1;
        for(int i = 0; i < weapons.Length; i++){
            //لاختيار أي سلاح نحتاج لقيمة المفتاح "1" مضافا إليها موقع السلاح في المصفوفة
            if(Input.GetKeyDown((KeyCode) keyCode + i)){
                currentWeapon = i;
                RefreshInHandValues();
            }
        }
    }

    void UpdateShooting(){
        //تم الضغط على زر الفأرة: قم بالضغط على الزناد
        if(Input.GetMouseButton(0)){
            weapons[currentWeapon].Fire();
        }
        //تم رفع الضغط عن زر الفأرة: أزل الإصبع عن الزناد
        if(Input.GetMouseButtonUp(0)){
            weapons[currentWeapon].ReleaseTrigger();
        }
        //تم الضغط على الزر الفأرة الأيمن: قم بإعادة تعبئة السلاح
        if(Input.GetMouseButtonDown(1)){
            weapons[currentWeapon].Reload();
        }
    }

    //Change weapon
    public void SetCurrentWeapon(int newIndex){
        weapons[currentWeapon].ReleaseTrigger();
        currentWeapon = newIndex;
        RefreshInHandValues();
    }

    void RefreshInHandValues(){
        foreach(GeneralWeapon gw in weapons){
```

```
                    //قيمة المتغير inHand يجب أن تكون true في حالة
                    //السلاح الذي يحمله اللاعب حاليا فقط
                    gw.inHand = weapons[currentWeapon] == gw;
               }
          }
}
```

السرد 84: البريمج الخاص بتبديل الأسلحة واستخدامها وإعادة تعبئتها

يجب أن تتم إضافة كافة الأسلحة التي قمنا بعملها إلى المصفوفة weapons وذلك حتى يتمكن البريمج من الوصول إليها وبالتالي تمكين اللاعب من التبديل بينها واستخدامها. القيمة المبدئية للمتغير initialWeapon هي 1-، مما يعني أنّ اللاعب لا يحمل أي سلاح بيده. بعد أن نقوم بإضافة الأسلحة الثلاث يمكننا تغيير هذه القيمة إلى 0 أو 1 أو 2 بحسب السلاح الذي نريده. المتغير الذي يحدد السلاح الحالي بيد اللاعب هو currentWeapon وهو متغير داخلي يمكن تغيير قيمته فقط عبر الدّالة SetCurrentWeapon(). هذا الأمر ضروري لضمان استدعاء الدّالة RefreshInHandValues() بعد كل عملية تبديل سلاح. أهمية الدّالة RefreshInHandValues() هو أنها تضمن وجود سلاح واحد فقط تكون فيه قيمة inHand هي true وبالتالي يمكنه الاستجابة للمدخلات. هذا السلاح الوحيد هو بطبيعة الحال الموجود داخل الموقع currentWeapon في مصفوفة الأسلحة. لقراءة المدخلات يتم استدعاء كل من UpdateShooting() و UpdateSwitching() في كل دورة تصيير.

بالحديث بالتفصيل عن الدّالة UpdateSwitching() نلاحظ أنها تقوم بفحص حالة مفاتيح الأرقام ابتداء من KeyCode.Alpha1 وهو المفتاح رقم 1 الذي تجده فوق الحرف Q أعلى يسار لوحة المفاتيح (لا ينطبق هذا على مفتاح 1 الأيمن الموجود بجوار الأسهم على لوح الأرقام). بعدها يتم تحويل قيمته من من المعدّد KeyCode إلى عدد صحيح int. هذا العدد لو أضفنا إليه 1 ومن ثم أعدنا تحويله إلى نوع المعدّد KeyCode فإننا نحصل على قيمة مساوية لـ KeyCode.Alpha2. هذه الحقيقة تساعدنا على فحص قيمة جميع الأرقام الممثلة بالأسلحة بشكل متسلسل عبر حلقة تكرارية مما يختصر علينا كتابة عدد كبير من جمل if-else لكل المفاتيح من KeyCode.Alpha1 إلى KeyCode.Alpha9. بالتالي فإنّ ضغط المفتاح 1 على لوحة المفاتيح سيحولنا للسلاح الموجود في الخانة صفر في المصفوفة weapons أي السلاح الأول والمفتاح 2 للسلاح الثاني وهكذا. على الجانب الآخر تقوم الدّالة UpdateShooting() بقراءة مدخلات أزرار الفأرة، حيث تستدعي الدّالة Fire() من السلاح الحالي عند الضغط على الزر الأيسر، كما تقوم باستدعاء ReleaseTrigger() حين إفلات هذا الزر. إضافة إلى ذلك فإنّ الضغط على زر الفأرة الأيمن سيؤدي لاستدعاء الدّالة Reload() من السلاح الحالي والتي تعمل على إعادة تعبئة السلاح. عند إعداد كائن اللاعب بشكل صحيح يجب أن يظهر في نافذة الخصائص كما في الشكل 90.



الشكل 90: كائن اللاعب حين إعداده بشكله النهائي

عند الانتهاء من إعداد كائن اللاعب بهذه الصورة تكون عملية الحصول على مدخلات الأسلحة قد اكتملت ويمكننا الانتقال للمخرجات. بما أنّ كلا من الرشاش والقناص سيعتمد على بث الأشعة في عملية التصويب، فمن المنطقي إعادة استخدام البريمج RaycastShooter (السرد 49 في الصفحة 145) وكل ما علينا فعله بعد إضافته لكل من السلاحين هو ضبط قيمه مثل المدى range وعدم الدقة inaccuracy و قوة الطلقة power. بعدها علينا أن نقوم بكتابة بريمج صغير هو عبارة عن "جسر" بين البريمجات الأخرى ومهمته استقبال الرسالة OnWeaponFire من البريمج GeneralWeapon واستدعاء الدّالة Shoot() من البريمج RaycastShooter بناء عليها. البريمج WeaponToRaycast الذي يقوم بهذه المهمة موضح في السرد 85.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(GeneralWeapon))]
[RequireComponent(typeof(RaycastShooter))]
public class WeaponToRaycast : MonoBehaviour {

	RaycastShooter shooter;

	void Start () {
		shooter = GetComponent<RaycastShooter>();
	}

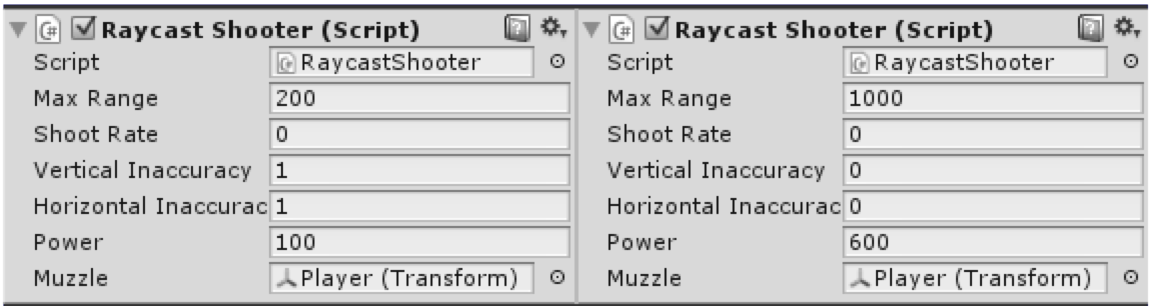
	void Update () {
	}

	void OnWeaponFire(){
		shooter.Shoot();
	}
```

```
}
```

السرد 85: بريمج بسيط يعمل كجسر بين GeneralWeapon و RaycastShooter

من البديهي أن يعتمد هذا البريمج على كل من RaycastShooter و GeneralWeapon كونه يعمل على الربط بينهما لا أكثر. عند إعداد كل من القناص والرشاش سيظهر البريمج RaycastShooter بشكله النهائي في نافذة الخصائص كما في الشكل 91.



الشكل 91: الإعدادان المختلفان للبريمج RaycastShooter على القناص (يمينا) وعلى الرشاش (يسارا)

يمكننا الآن العودة إلى قالب الوحدة البنائية الذي قمنا بنسخه وإجراء التعديلات اللازمة عليه. بالإضافة لكون هذه الوحدات البنائية قابلة للهدم، عليها أن تتأثر بطلقات البث الإشعاعي التي يطلقها RaycastShooter. علينا أولا أن نتأكد من أن الطلقات تحدث ثقوبا في هذه الوحدات البنائية وذلك عن طريق إضافة البريمج BulletHoleMaker (السرد 52 في الصفحة 150) إلى القالب الجديد الذي أسميناه ShootableBrick ونحدد قالب الثقب الذي سبق واستخدمناه. علينا أيضا أن نقوم بإزالة البريمج MouseExploder لتجنب حدوث انفجار عند كل نقرة بالفأرة على أي وحدة بنائية.

لإزالة مكوّن أو بريمج من كائن ما قم بالضغط بالفأرة على رمز الترس في أعلى يسار المكوّن ومن ثم اختر Remove Component

البريمج التالي الذي ينبغي أن نضيفه للقالب هو BulletForceReceiver (السرد 53في الصفحة 152) والذي يسمح لطلقات كل من الرشاش والقناص أن يطبقا قوة دفع فيزيائية على الأجسام التي تصيبها. من المهم ملاحظة أن هذا التأثير على الوحدة البنائية غير ممكن بشكلها الحالي؛ والسبب هو أنها تحتوي على البريمج Destructible والذي يقوم بتجميد الجسم الصلب الخاص بها ويمع تأثير القوى الخارجية عليها، مما يعني أن البريمج BulletForceReceiver لن يكون ذو تأثير قبل أن يتم هدم الوحدة البنائية. لحل هذه المشكلة سنحتاج لبريمج يقوم بهدم الوحدة البنائية حين إصابتها بإطلاق نار ببث شعاعي (أي عند استقبال الرسالة OnRaycastHit) ذو قوة كافية لهدمها. هذا البريمج هو DestructOnHitDamage وهو موضح في السرد 86.

```
using UnityEngine;
```

```
using System.Collections;

[RequireComponent(typeof(Destructible))]
public class DestructOnHitDamage : MonoBehaviour {

	//الحد الأدنى لقوة الإصابة القادرة على هدم الوحدة البنائية
	public float destructionDamage = 250;

	Destructible dest;

	void Start () {
		dest = GetComponent<Destructible>();
	}

	void Update () {

	}

	void OnRaycastHit(RaycastHit hit){
		//distance تذكر أن قوة الإصابة مخزنة داخل المتغير
		//destructionDamage إذا كانت قيمتها أكبر من
		//نقوم بهدم الوحدة البنائية
		if(hit.distance > destructionDamage){
			dest.Destruct();
		}
	}
}
```

السرد 86: البريمج الخاص باستقبال إصابة بث الأشعة وهدم الوحدة البنائية المصابة بناء على قوة الإصابة

كل ما علينا فعله الآن هو تحديد مقدار القوة اللازمة لهدم الوحدة البنائية. عند استدعاء الدّالة ()Destruct فإننا نقوم بإزالة كافة القيود عن حركة الوحدة البنائية مما يسمح لقوة دفع الرصاصة التي يقوم البريمج BulletForceReceiver بالتأثير على موقع الوحدة البنائية ودورانها.

آخر ما علينا التعامل معه في موضوع مخرجات السلاح هو قذائف RPG. سنحتاج في هذه الحالة إلى قالب للقذيفة والتي سيتم إطلاقها عند استخدام سلاح القاذفة. في حال اصطدام هذه القذيفة بأحد الجدران فستقوم بإحداث انفجار وتدميره كليا أو جزئيا. سنحتاج لجزأين في عملية إطلاق القذيفة: القذيفة نفسها بالإضافة إلى بريمج يستجيب للرسالة OnWeaponFire عن طريق إطلاق هذه القذيفة في اتجاه التصويب الحالي. لنبدأ مع البريمج RPG والذي يعمل على إطلاق القذيفة والذي يجب أن نضيفه لكائن سلاح RPG. هذا البريمج موضح في السرد 87.

```
using UnityEngine;
using System.Collections;

public class RPG : MonoBehaviour {

	//قالب القذيفة التي سيتم إطلاقها
	public GameObject rocketPrefab;

```

```
	void Start () {

	}

	void Update () {

	}

	//قم باستقبال الرسالة وإطلاق القذيفة بناء على ذلك
	//سنعطي القذيفة موقع ودورانا أوليين مساويان للموقع والدوران الخاصين
	//بكائن القاذفة نفسها
	void OnWeaponFire(){
		GameObject rocket = (GameObject)Instantiate(rocketPrefab);
		rocket.transform.position = transform.position;
		rocket.transform.rotation = transform.rotation;
	}
}
```

السرد 87: بريمج قاذفة RPG

سنستخدم لعمل القذيفة كرة سنقوم بتمديدها على محورها المحلي z بحيث تصبح إهليجية الشكل (أشبه بكرة القدم الأمريكية). يمكننا مثلا استخدام الأبعاد (0.2, 0.2, 0.75) ومن ثم نعطي الكائن أي إكساء مناسب. بعد ذلك سنقوم ببناء قالب من هذا الكائن ونربط هذا القالب بالمتغير rocketPrefab في البريمج RPG. سيحتاج هذا القالب بطبيعة الحال لعدد من المكونات والبريمجات حتى يصبح سلوكه كقذيفة صحيحا. بداية سنحتاج لمكوّن الجسم الصلب rigid body وذلك لنكون قادرين على تحريكه بقوة دافعة واكتشاف تصادمه مع الكائنات الأخرى. قبل الخوض في تفاصيل المكوّنات الأخرى علينا أن نتأمل ما تقوم به القذيفة: أولا يتم إطلاقها بقوة دافعة من القاذفة، ثانيا تمتلك القدرة على الانفجار، ثالثا تقوم بتفجير الهدف الذي تصيبه، رابعا تُدمر عند التصادم التصادم. لكل واحد من هذه الخصائص الأربعة سنحتاج بريمجا مختلفا، ولتكن البداية مع الإطلاق والحركة والتي يتولاها البريمج RPGRocket الموضح في السرد 88.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Rigidbody))]
public class RPGRocket : MonoBehaviour {

	//القوة الدافعة عند الإطلاق
	public float launchForce = 100;

	//عدد الثواني التي تبقى خلالها القذيفة موجودة في المشهد
	//في حال لم تصب أي هدف
	public float lifeTime = 7;

	//متغير داخلي لحساب وقت الإبقاء على القذيفة
	float launchTime;

	//متغير داخلي لتتبع حالة القذيفة
```

```
    //هذا المتغير مهم لمنع اكتشاف التصادم بين كائن القذيفة بعد اصطدامها
    //والأجزاء التي ستنتج عنها عند تحطمها
    bool destroyed = false;

    void Start () {
        rigidbody.AddForce(transform.forward * launchForce,
                                          ForceMode.VelocityChange);

        launchTime = Time.time;
    }

    void Update () {
        if(!destroyed && Time.time - launchTime > lifeTime){
            Destroy(gameObject);
        }
    }

    void OnCollisionEnter(Collision col){
        if(!destroyed){
            destroyed = true;
            //قم بإعلام البريمجات الأخرى بحدوث التصادم مع كائن آخر
            //وأرفق مع الرسالة مرجعا لهذا الكائن
            SendMessage("OnRocketHit",
                col.collider,
                SendMessageOptions.DontRequireReceiver);

            //قم بتدمير كائن الصاروخ
            Destroy(gameObject);
        }
    }
}
```

السرد 88: البريمج الخاص بإطلاق قذيفة RPG واكتشاف تصادمها مع الكائنات الأخرى

يعطي هذا البريمج عند بدايته قوة دافعة لإطلاق القذيفة، بحيث تكون قوة الإطلاق نحو الأمام. بعد ذلك يبدأ البريمج في حساب عمر القذيفة الذي تم تحديده قبل تدميرها. على أي حال فإن اصطدام القذيفة بهدف ما سيؤدي إلى تدميرها حتى لو لم ينقضي العمر المحدد لها، لكن هذا التدمير لن يتم قبل إرسال الرسالة OnRocketHit والتي تخبر البريمجات الأخرى بحدوث التصادم وتحمل إليها مرجعا للجسم الذي تم التصادم معه. لاحظ أنّه من الممكن أن تصطدم القذيفة بأكثر من جسم في نفس الإطار، كما يمكنها أن تتصادم مع حطامها الذي سنقوم بإنشائه كما سنرى بعد قليل. من أجل ذلك نحتاج لاستخدام متغير الحالة الداخلي destroyed حتى نضمن إرسال الرسالة OnRocketHit مرة واحدة فقط. بالعودة لموضوع الحطام، يمكننا إعادة استخدام البريمج Breakable الذي سبق وكتبناه (السرد 59 في الصفحة 170) وسنستخدم قالبا خاصا بالقذيفة ليمثل قطع حطامها. فيما يتعلق بالمتغير explosionPower، علينا هذه المرة أن نستخدم قيمة عالية نسبيا (1000 مثلا) بحيث تمثل انفجارا قويا تحدثه القذيفة حين اصطدامها ويؤثر إلى تناثر قطع الحطام بعيدا عن بعضها البعض.

الخطوة التالية هي الربط بين تصادم القذيفة مع الهدف وتدميرها، من أجل هذا نحتاج لبريمج صغير

هدفه ربط البريمج RPGRocket مع البريمج Breakable. ما سيقوم به هذا البريمج هو استقبال الرسالة OnRocketHit من RPGRocket وإرسال الرسالة Break (أو استدعاء الدّالة ()Break مباشرة) إلى البريمج Breakable. البريمج هو BreakOnRocketHit وهو موضح في السرد 89.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(RPGRocket))]
[RequireComponent(typeof(Breakable))]
public class BreakOnRocketHit : MonoBehaviour {

	void Start () {

	}

	void Update () {

	}

	void OnRocketHit(Collider hitObject){
		GetComponent<Breakable>().Break();
	}
}
```

السرد 89: بريمج يربط RPGRocket و Breakable بحيث يؤدي لتدمير القذيفة مباشرة عند اصطدامها مع الهدف

البريمج الأخير الذي سنضيفه لقالب القذيفة هو المادة المتفجرة التي ستقوم بتدمير الهدف، عند استقبال الرسالة OnRocketHit من البريمج RPGRocket علينا أن نقوم بالبحث عن الوحدات البنائية الموجودة ضمن مدى الانفجار وهدمها بحيث تصبح تحت تأثير قوة الانفجار التي سنضيفها لها. هاتان الخطوات يقوم بهما البريمج DestructOnRocketHit والذي يوضحه السرد.

```
using UnityEngine;
using System.Collections;

public class DestructOnRocketHit : MonoBehaviour {

	//نصف قطر محيط تأثير الانفجار
	public float explosionRadius = 3;

	//قوة الانفجار
	public float explosionForce = 50000;

	void Start () {

	}

	void Update () {

	}

	void OnRocketHit(Collider target){
```

```
            //قم بهدم جميع الوحدات البنائية الواقعة ضمن محيط الانفجار
            //وإضافة قوة الانفجار لها بعد ذلك
            Destructible[] all = FindObjectsOfType<Destructible>();

            Vector3 explosionPos = transform.position;

            foreach(Destructible dest in all){
                  if(Vector3.Distance
                        (explosionPos, dest.transform.position)
                                          < explosionRadius){

                        dest.Destruct();

                        dest.rigidbody.
                              AddExplosionForce(explosionForce,
                                          explosionPos,
                                          explosionRadius);
                  }
            }
      }
}
```

السرد 90: بريمج لهدم وتفجير الوحدات البنائية المحيطة بمكان اصطدام قذيفة RPG

لعلك لاحظت الشبه الكبير بين البريمجين MouseExploder (السرد 57 في الصفحة 165) و هذا البريمج، الفرق هو أنّ MouseExploder يعتمد على موقع مؤشر الفأرة كموقع للانفجار بينما DestructOnRocketHit يعتمد على موقع اصطدام القذيفة.

بهذا تكون جميع الأسلحة جاهزة للاستخدام ويمكنك تجربتها في المشهد. يمكن إطلاق النار من الأسلحة عن طريق زر الفأرة الأيسر كما يمكن التبديل بينها باستخدام مفاتيح الأرقام 1 و 2 و 3. آخر ما يتوجب علينا فعله الآن هو تفعيل آلية عرض كمية الذخيرة ومقدار التقدم في عملية إعادة التعبئة للأسلحة الثلاث. بالعودة إلى الشكل 88 يمكنك ملاحظة وجود كائن نص يحمل القيمة (amm) إلى جانب كل اسم من أسماء الأسلحة الثلاث. سنستخدم هذه النصوص لعرض معلومات عن كل سلاح، فمثلا سنعرض النص (XXX) أمام كل سلاح لا يحمله اللاعب حاليا للدلالة على أنه ليس قيد الاستخدام في هذه اللحظة، بالتالي يكون اللاعب دوما على علم بالسلاح الذي يحمله. أمّا إذا كان السلاح بيد اللاعب فإننا سنعرض كمية الذخيرة المتبقية مستخدمين التنسيق A/B حيث أن A هو كمية الذخيرة المتبقية في المخزن الحالي و B هو عدد المخازن المتبقية. أمّا إذا كان السلاح قيد إعادة التعبئة فإنّ تقدم العملية سيظهر على شكل نسبة مئوية.

للتحكم بعرض حالة السلاح سنضيف البريمج AmmoDispay الموضح في السرد 91 إلى كل واحد من الأسلحة الثلاث التي قمنا بإنشائها.

```
using UnityEngine;
using System.Collections;
```

```
[RequireComponent(typeof(GeneralWeapon))]
public class AmmoDisplay : MonoBehaviour {

	//كائن النص الذي سيتم عرض المعلومات عليه
	public TextMesh display;

	//مرجع لبريمج السلاح الذي سنستخدمه كمصدر للمعلومات المعروضة
	GeneralWeapon weapon;

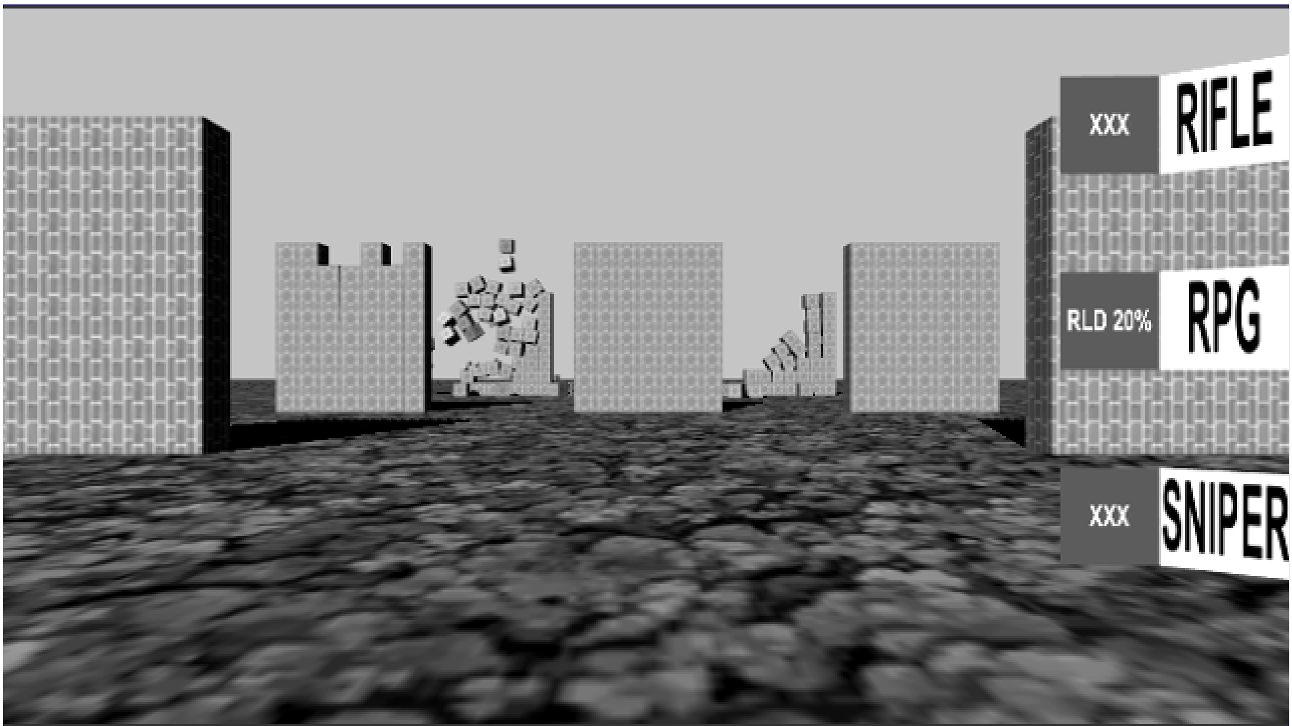
	void Start () {
		weapon = GetComponent<GeneralWeapon>();
	}

	void LateUpdate () {
		//لا تعرض أي معلومات ما لم يكن اللاعب يحمل السلاح
		if(!weapon.inHand){
			display.text = "XXX";
			return;
		}

		float reloadProgress = weapon.GetReloadProgress();
		//قم بعرض كمية الذخيرة المتبقية في المخزن وعدد المخازن المتوفرة
		if(reloadProgress == 0){
			display.text = weapon.magazineSize + "/" +
						weapon.magazineCapacity + " (x" +
							weapon.magazineCount + ")";
		} else {
			//في حالة إعادة التعبئة قم بعرض التقدم في هذه العملية
			int progress = (int)(reloadProgress * 100);
			display.text = "RLD " + progress + "%";
		}
	}
}
```

السرد 91: البريمج الخاص بعرض حالة السلاح وكمية الذخيرة المتبقية على شكل نص

لاحظ أننا نستخدم القيمة المرجعة من الدّالة ()GetReloadProgress لمعرفة ما إذا كان السلاح قيد إعادة التعبئة أم لا، وذلك اعتمادا على كون القيمة تساوي صفرا أو أكبر من ذلك؛ حيث أن القيمة صفر تعني أنّ السلاح جاهز للاستخدام ولا تتم حاليا إعادة تعبئته. في هذه الحالة سنقوم بعرض الذخيرة المتبقية كما سبق وذكرنا، أما في حالة إرجاع قيمة أخرى غير الصفر من ()GetReloadProgress فإنّ هذه القيمة تمثل مقدار التقدم في إعادة التعبئة على شكل قيمة بين صفر و 1، والتي نقوم بضربها بالرقم 100 لنحصل على رقم أسهل للعرض يكون بين صفر و 100. الشكل 92 يعرض المشهد بصورته النهائية أثناء استخدام أحد الأسلحة، يمكن أيضا الاطلاع على النتيجة في المشهد scene22 في المشروع المرفق.



الشكل 92: الشكل النهائي لمشهد تجربة تبديل الأسلحة

تمارين

1. قم بإنشاء باب دوار مستخدما مكوّن المفصل الرزي Hinge Joint ومن ثم قم بقفله بقفل يتطلب من اللاعب جمع مفتاحين مختلفين حتى يتمكن من فتحه. يمكنك استخدام البريمجات المشروحة والتعديل عليها بما يناسب، كما يمكنك كتابة بريمجاتك الخاصة.

2. قم بعمل لغز فك قفل يعتمد على دفع اللاعب لثلاث صناديق ووضعها في ثلاث مواقع مختلفة. حين يتم وضع كل صندوق في مكانه الصحيح على الأرض يفتح الباب تلقائيا.

3. قم بإضافة بريمج للعبة التي أنشأناها في الفصل الثالث بحيث يقوم بشكل عشوائي على فترات زمنية بإلقاء صندوق مساعدات للاعب. إذا التقط اللاعب هذا الصندوق عبر لمسه فإنّه يزيد صحته بمقدار معين (20 مثلا). راعي أيضا أنّه في حال أصابت إحدى الطلقات صندوق المساعدات قبل أن يصله اللاعب فإنه يُدمّر مباشرة.

4. قم بإضافة سلاح قنبلة يدوية للمثال في الفصل الرابع. عندما يؤشر اللاعب على الهدف بالفأرة ويطلق النار يجب أن يتم رمي القنبلة في ذلك المكان كما يجب أن تنفجر بعد 7 ثوان من رميها. عند انفجارها يجب أن تؤثر على جميع الوحدات البنائية ضمن مجال تأثيرها.